Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Departamento de Informática

Refinamento de Diagramas de Classes: Análise e Verificação

Por

Ana Cristina Martins Ferreira

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboapara obtenção do grau de Mestre em Engenharia Informática.

Orientadora

Profª. Doutora Carla Ferreira

Lisboa

2010

Agradecimentos

Deixo expresso o meu eterno e sincero agradecimento à minha orientadora Profª Carla Ferreira pela sua orientação, confiança e amizade, a qual nunca esquecerei.

Também o meu agradecimento a todos que deram sugestões e ideias ao longo deste caminho.

Dou muito valor a todos os documentos que tenho lido pois eles fizeram-me deslumbrar e ter em conta os grandes desafios que existem, mas também as soluções únicas que ofereciam. O meu trabalho baseia-se nas suas contribuições.

Por último ainda mais importante, gostaria de agradecer à minha família, pais e irmãos, pelo seu apoio, amor incondicional e incentivo na busca dos meus objectivos e por tudo que me proporcionaram, tendo sempre presente que o sonho é uma constante da vida.

Nº do aluno: 26340

Nome: Ana Cristina Martins Ferreira

Título da dissertação:

Refinamento de Diagramas de Classes: Análise e Verificação

Palavras-Chave:

- Diagramas de classes
- UML (Unified Modelling Language)
- OCL (Object Constraint Language)
- Alloy e Alloy Analyzer
- Transformações de refinamento

Keywords:

- Class diagrams
- UML (Unified Modelling Language)
- OCL (Object Constraint Language)
- Alloy and Alloy Analyzer
- Refinement transformations

Resumo

A qualidade do Software foi sempre uma das grandes preocupações das empresas de desenvolvimento de software. O suporte à constante necessidade de actualização e adaptações dos sistemas é essencial ao sucesso dos mesmos.

O paradigma orientado a objectos centraliza uma grande parte dos seus esforços, na criação de sistemas de software bem planeados, robustos, modificáveis e, sempre que possível, reutilizáveis.

O UML (Unified Modelling Language) é uma linguagem de modulação visual, complementada pela linguagem de especificação formal de restrições OCL (Object Constraint Language). O OCL permite aumentar a expressividade dos diagramas, mas não consegue colmatar totalmente a informalidade do UML.

O refinamento baseia-se na premissa de que temos uma dada especificação e através de um conjunto de regras bem-formadas podemos obter uma segunda especificação aperfeiçoada, em que o comportamento inicialmente observado é mantido.

O Alloy é uma linguagem de especificação formal, orientada a objectos, direccionada para a criação de micro-modelos, que nos possibilita a análise e verificação formal, através da ferramenta associada Alloy Analyzer.

Com o objectivo de verificar a correcção das transformações de refinamento de diagramas de classe do UML, propomos a definição de um conjunto de regras de refinamento. Com base nessas regras e nos modelos (modelo original e concreto), propomos usar a ferramenta Alloy Analyzer para a verificação formal automática da correcção do refinamento.

Abstract

Software quality has always been a major concern for software development teams. Support for the constant need to update and adapt software systems is essential to the success of those systems.

The object oriented paradigm centralizes a large part of its efforts in creating software systems that are well planned, robust, modifiable, and, where possible, reusable.

UML (Unified Modeling Language) is a visual modeling language that includes OCL (Object Constraint Language), a formal specification language for expressing constraints over objects. OCL increases the expressivity of UML diagrams, but it does not overcome completely the informality of UML.

Refinement is based on the premise that given a specification and through a set of well-formed rules we can attain a second more detailed specification, where the original behavior is ensured. Alloy is a formal specification language, object-oriented, suitable for the defining micro-models. This language has an associated tool, the Alloy Analyzer, which supports automatic analysis and formal verification. With the aim of verifying refinement transformations of UML class diagrams, we intend to define a set of refinement rules. Based on those rules and on both abstract and concrete models, we propose to use the Alloy Analyzer for automatic formal verification of the refinement correction.

Índice

Índice de Tabelas

Índice de Figuras

1 Introdução

Este capítulo destina-se à apresentação dos conteúdos que deram origem à dissertação de Mestrado "Refinamento de Diagrama de Classes: Análise e Verificação". Neste âmbito, apresentamos a motivação para este trabalho, os problemas encontrados, os objectivos que almejamos cumprir e o contributo que ambicionamos puder dar nesta área.

1.1 Motivação

A Engenharia do Software é uma área da computação direccionada para a especificação, desenvolvimento e manutenção de sistemas, que utiliza diferentes tipos de tecnologias e práticas, com o objectivo de promover a organização, produtividade e qualidade do Software.

O desenvolvimento de Software revela-se ser uma tarefa árdua, complexa, difícil, e consumidora de vários tipos de recursos. Nos nossos dias, a qualidade do Software tem-se imposto como um factor determinante para o seu sucesso.

Em qualquer que seja o domínio da indústria exige-se, cada vez mais, capacidade de inovação e de adaptação, um bom desempenho, fiabilidade e robustez nos produtos.

A evolução é um feito inato e essencial, pois é uma consequência natural e ambicionável. Esta permite que sejam feitos melhoramentos, adaptações e reutilizações no Software.

No entanto, a maioria das estruturas não comportam facilmente as adaptações, sendo necessário facultar formas de reorganização do Software, para que este se torne cada vez mais adaptável, consistente, reutilizável e fiável.

Com o intuito de simplificar o processo de desenvolvimento muitos especialistas recorrem à estratégia "dividir para conquistar". Esta permite dividir os modelos em modelos mais pequenos, de forma a torná-los mais compreensíveis e intuitivos.

A desvantagem desta estratégia reside num aumento substancial de novos modelos mais pequenos, e no facto destes não poderem ser analisados de forma individual porque, na prática, a maioria está dependente do modelo global. Nesta perspectiva, é fulcral a correcta manipulação dos modelos.

Assumimos que um modelo é uma representação fragmentada do mundo real de um dado domínio do problema, onde se incidem as tarefas de modelação e construção da informação de um sistema, segundo uma determinada estrutura de conceitos [39]. Este deve ser simples e pragmático, apresentando uma visão completa do seu âmbito. No entanto, esta representação torna-se limitada quando se pretende apresentar toda a informação necessária, de forma detalhada, sobre o sistema. Um dos mecanismos chave que permite capturar estes detalhes é a abstracção.

A abstracção é um mecanismo que permite eliminar complexidade centrando-se somente nos aspectos essenciais e relevantes para a solução em análise. Desta forma, obtemos modelos simples e pragmáticos.

O UML [1] (Unified Modeling Language) é uma linguagem de modulação gráfica, não formal, orientada a objectos, adoptada pela OMG (Object Management Group), resultado de um esforço conjunto de várias arquitecturas gráficas, com finalidades precisas:

- Reflectir as melhores práticas da indústria de Software; e
- Desmistificar a complexidade do processo de modelação.

Esta linguagem tem a capacidade de capturar aspectos de um sistema segundo diferentes visões. Estas vistas são descritas graficamente por diferentes tipos de diagramas, que no seu conjunto produzem uma descrição completa do sistema. No entanto, se não houver auxílio de outros mecanismos, as múltiplas vistas definidas podem ser incoerentes, devido ao informalismo, ainda existente no UML, que permite que uma vista tenha várias interpretações possíveis.

Ao nível empresarial, adopção do UML tem sido satisfatória devido ao seu cariz apelativo e intuitivo, que possibilita que uma equipa de desenvolvimento consiga rapidamente integrar-se e conhecer, de forma geral, o âmbito e desenho de todo o sistema, em qualquer fase do processo de desenvolvimento de software.

No entanto, quando estamos perante uma situação que requeira algum tipo de alteração no sistema, normalmente, temos que reestruturar os diagramas e ter em consideração que as relações estabelecidas são respeitadas. Neste âmbito, somos confrontados com a problemática da consistência e da preservação da semântica das transformações

realizadas. Isto porque depois de realizar qualquer tipo de transformação há que verificar se o seu comportamento original observado é mantido.

Na questão da preservação semântica, os métodos formais são um artefacto capaz de providenciar rigor permitindo analisar e detectar as inconsistências, que de outra forma seriam de difícil percepção, assim como de difícil resolução.

Nem sempre os benefícios que advêm dos métodos formais têm tido a adesão esperada por parte da indústria. Podemos especular que este facto se deve, não só à postura de determinados analistas

> *'acharem que as soluções que recorram a estes requerem demasiados recursos, que posteriormente não são rentabilizados [10]'*

mas também pela falta de regulamentação pelas entidades responsáveis, assim como a falta de promoção pela própria comunidade cientifica.

No entanto, é consensual que a sua utilização produz resultados bastante satisfatórios e que é um excelente recurso, principalmente quando existe suporte de análise e verificação automática.

A técnica de refinamento é um dos mecanismos usados quando estamos perante situações de evolução de um sistema. Nesta óptica, definimos o refinamento como pequenas transformações capazes de introduzir informações úteis e precisas que facilitam as tomadas de decisões ao nível do desenho, num modelo, permitindo assim que este seja o mais próximo possível do modelo de implementação.

Os refinamentos, em geral, são feitos de forma gradual, produzindo modelos cada vez mais próximos da sua implementação.

1.2 Descrição do Problema

Um dos problemas que a maioria das organizações enfrenta é a rápida desactualização dos seus sistemas. Geralmente, estas estruturas não conseguem adaptar-se facilmente às alterações que são necessárias para se manterem actualizadas. Assim sendo, é necessário proporcionar alicerces capazes de avaliar as alterações num sistema, minimizando de forma eficiente e eficaz os prejuízos que possam daí advir.

Como referido, o UML é uma linguagem de modulação gráfica, composta por vários tipos de diagramas: diagramas estáticos e diagramas comportamentais. No caso dos diagramas estáticos destacamos os diagramas de classes.

Neste trabalho vamos considerar um modelo como um diagrama de classes UML. O conceito de modelo é um instrumento que atravessa os mais diferentes tipos de

linguagens, sejam elas de programação, ou de modelação, entre outras, enquanto os diagramas de classes são instrumentos específicos da linguagem UML.

A construção de um novo modelo proveniente de uma alteração obriga-nos a verificar se este está consistente relativamente ao seu antecessor. Dado o cariz desta tarefa, esta deve ser feita de forma automática, o que facilita a correcção dos modelos.

No caso do UML, uma das suas limitações é a falta de formalismo para a colmatar, é complementado pela linguagem de especificação formal OCL [16]. Esta linguagem permite definir restrições, baseando-se na utilização da teoria dos conjuntos e lógica de primeira ordem.

O UML não tem suporte automático de análise de refinamentos. No entanto, esta situação pode ser suprimida com a tradução dos diagramas por uma linguagem de especificação formal. Geralmente, este processo de tradução recorre ao mapeamento directo de cada elemento da linguagem UML para a linguagem formal destino.

O Alloy é uma linguagem de especificação formal, orientada a objectos, do tipo declarativo, permitindo expressar restrições estruturais complexas, com suporte de análise automática. Assim, neste trabalho utilizaremos a linguagem de especificação formal Alloy, e a sua ferramenta de análise automática para analisar a consistência do refinamento realizado.

No âmbito deste trabalho, propomos definir um conjunto de regras de refinamento para os diagramas de classes UML que possibilitem uma análise e verificação automática da validade de um refinamento.

Nesta dissertação teremos a seguinte envolvente problemática:

- Falta de ferramentas para análise e verificação automática, de modelos gráficos;
- Estender regras de refinamento de diagramas de classes já propostas noutros trabalhos; e
- Mapeamento de diagramas de classes UML para modelos Alloy.

Para a análise da validade do refinamento entre dois modelos é necessário ver:

- Consistência das relações entre classes;
- Equivalência entre classes.

Através do Alloy Analyzer, é gerada uma resposta que indica qual o estado do modelo refinado relativamente ao modelo original, isto é, se modelo refinado é ou não equivalente ao modelo original.

1.3 Contribuições

Sempre que um sistema entra em produção, deve ser submetido a testes rigorosos, de forma a que estes possam garantir que são cumpridos os requisitos funcionais e não-funcionais. Os testes permitem detectar um conjunto de falhas e erros. Esta tarefa revela-se ser um fardo pesado e bastante aglutinador de recursos.

A utilização dos diagramas de classes com suporte do Alloy, permite a detecção de algumas destas falhas numa fase inicial do projecto, permitindo que desta forma se rentabilize efectivamente os recursos, nomeadamente o recurso de tempo e esforço da fase de testes, facultando a automatização da análise.

Nesta óptica e partindo da premissa que quanto mais formal for um modelo maior é a capacidade de providenciar suporte automatizado das diferentes actividades de desenvolvimento, propomos:

- Definição de um conjunto de regras de refinamento entre diagramas de classes UML, passíveis de análise automática, e
- Definição de um mapeamento de UML para Alloy.

Os diagramas de classes são traduzidos para a linguagem de especificação formal Alloy, onde o resultado obtido é submetido à sua ferramenta de suporte de análise, o Alloy Analyzer.

No cerne desta tese, temos o mecanismo de refinamento que permite adicionar informação a um modelo aumentando o seu grau de detalhe, assim como analisar e verificar situações inexploradas que ajudam à prevenção de erros subtis. Neste contexto, definimos um conjunto de regras de refinamento para os diagramas de classes UML cujo intuito é garantir a preservação semântica, que possibilitem uma análise e verificação automática da validade de um refinamento. Estas regras permitem averiguar se os modelos são equivalentes, ou não. Logo, podemos inferir se estes são, ou não, consistentes.

Sendo um diagrama aperfeiçoado um melhoramento de um diagrama original, deve existir uma forma directa, manual ou automática, que nos permita inferir directamente toda a informação alterada. Neste contexto, utilizamos o conceito de rastreabilidade como guia que permite a reconstituição completa das várias etapas do desenvolvimento.

Na Figura 1 apresentamos, de forma esquematizada, a metodologia do nosso trabalho.

Os diagramas de classes "*original*" e "*aperfeiçoado*" juntamente com a tabela de rastreabilidade são submetidos às regras de transformação UML para Alloy que

juntamente com as regras de refinamento servirão de dados de entrada à verificação formal dos modelos.

Para automatizar a análise do refinamento, este tem de ser desenvolvido através de modelos formais..

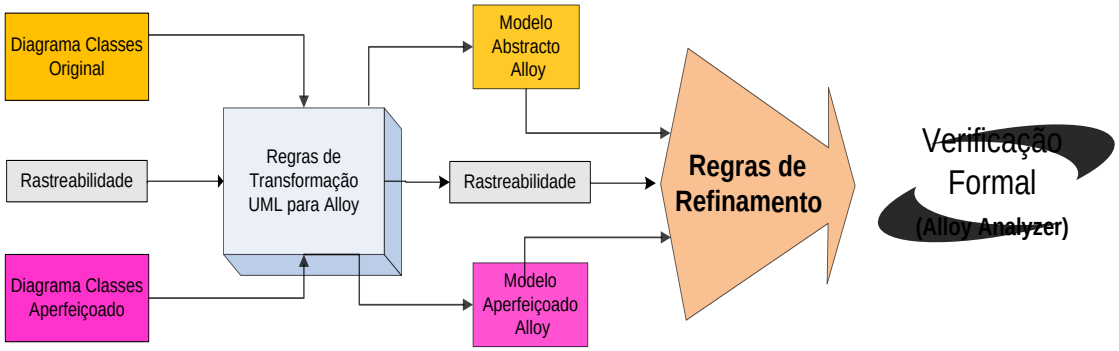


Figura 1: Metodologia proposta

Como referido, já existem outros mapeamentos de UML para Alloy [17]. No entanto, neste trabalho iremos usar uma abordagem diferente.

1.4 Estrutura do Documento

Finalizamos este capítulo com a apresentação da organização deste documento. Este está dividido nos capítulos descritos de seguida:

No capítulo 2 apresentamos uma descrição da linguagem de modelação UML, dos elementos básicos dos diagramas de classes e dos diagramas de actividades, a linguagem de restrições OCL e finalizamos com um metamodelo genérico dos diagramas de classes UML. No capítulo 3 apresentamos as abordagens existentes relativas ao estudo da técnica de refinamento, das linguagens de especificação formal, assim como dos métodos formais. No capítulo 4 apresentamos os princípios básicos, a lógica e modelação em Alloy, assim como descrevemos o seu analisador, o Alloy Analyzer. No capítulo 5 descrevemos o mapeamento dos elementos de um diagrama de classes UML para um modelo Alloy, apresentamos a estrutura dos módulos do sistema de verificação, descrevemos o metamodelo de transformação de um diagrama de classes UML para modelo Alloy, assim como apresentamos a estrutura da rastreabilidade a utilizar. No capítulo 6 descrevemos o sistema de regras de refinamento proposto. No capítulo 7 apresentamos o caso de estudo Arena. No capítulo 8 finalizamos com as conclusões e trabalho futuro.

2 Estado de Arte

No processo de desenvolvimento do software, a correcta modelação das aplicações deve-se apoiar em mecanismos precisos de descrição, assim como nas suas formas de comunicação. Um dos desafios desta conjuntura é o estabelecimento do equilíbrio entre a simplicidade (que favorece a comunicação) e formalização (que favorece a precisão), de um modelo.

Os modelos são artefactos que permitem a representação abstracta das entidades a modelar. Estes podem ter diferentes formas de representar os seus relacionamentos, as suas propriedades e os problemas a que se propõem resolver. É irrealista pretender construir um único modelo, esperando que este possa revelar todos os detalhes de um sistema.

Neste capítulo, analisamos algumas técnicas que permitem a reestruturação de modelos estruturais. No âmbito do nosso trabalho, um modelo estrutural é descrito por um diagrama de classes.

Após uma modificação ou transformação, é essencial verificar se o modelo refinado está consistente, relativamente ao modelo original, isto porque o modelo refinado tem de manter o comportamento observado.

Ainda neste capítulo, serão analisados os conceitos de refinamento, algumas abordagens que usam refinamentos, técnicas de verificação formal e transformações de refinamento. Cada técnica apresentada pode diferir nos métodos usados, nos procedimentos empregues, e nas ferramentas utilizadas de suporte à modelação, porém, todas elas se regem pela mesma premissa: “todas as transformações a realizar em modelos ou em diagramas de classes, podem ser realizadas em diferentes nas fases da sua evolução ou nas diferentes etapas do ciclo de vida de um sistema, mas o comportamento original observado tem de ser mantido”.

A modelação é um meio que privilegia a capturar de ideias, de relacionamentos, de decisões e de requisitos, sob uma notação formal bem-definida, podendo ser aplicada a diferentes contextos de trabalho.

2.1 UML

Com referido, o UML é uma linguagem vocacionada para a especificação, construção e documentação de artefactos de um sistema [39]. Esta tem uma forte aceitação, tanto no meio académico como no meio industrial, devido ao seu vasto âmbito de aplicabilidade em metodologias de desenvolvimento ou ferramentas de apoio.

O UML apresenta as seguintes características:

- Independência do domínio de aplicação;
- Independência do proceso ou metodologia de desenvolvimento;
- Facilita mecanismos de extenção.

O UML permite capturar diferentes aspectos de um sistema através de diferentes tipos visões. Os sistemas são compostos por diferentes níveis de abstrações. Uma visão pode ser descrita por um ou mais diagramas que evidenciam aspectos de um sistema.

Existem os seguintes tipos de visões: a visão estrutural, a visão funcional e a visão dinâmica.

Na visão estrutural, os diagramas permitem representar toda estrutura de um sistema. Nesta visão temos os seguintes diagramas:

- Diagrama de Pacotes – apresenta a organização, dividida em agrupamentos lógicos, dos elementos de um pacote e suas respectivas dependências;
- Diagrama de Classes – apresenta a estrutura estática de um sistema, definindo todas as classes que o constituem, assim como os relacionamentos entre as diferentes classes;
- Diagrama de Objectos – apresenta as instâncias de uma classe, num determinado momento da execução, indicando assim o estado nesse ponto de execução;
- Diagrama da Estrutura Composta – apresenta a cooperação de um conjunto de entidades na execução de uma determinada tarefa;
- Diagrama de Componentes – apresenta todos componentes de um sistema e respectivas dependências;
- Diagrama de Instalação – apresenta a configuração e arquitectura de um sistema.

Na visão funcional, os diagramas permitem descrever todas as funcionalidades de um sistema. Nesta visão temos os seguintes diagramas:

- Diagrama de Casos de Utilização – apresenta um conjunto de cenários na óptica do utilizador, que devem ser utilizados no processo de modelação do sistema, servindo de base à construção dos restantes diagramas;
- Diagrama de Actividades – apresenta a execução de um conjunto de acções que possibilitam representar um processo operacional, de um sistema.

Na visão dinâmica, os diagramas permitem representar todos aspectos sujeitos a alterações e a forma como os objectos interagem entre si, nomeadamente o fluxo de mensagens e a movimentação física dos componentes, de um sistema. Nesta visão, temos os seguintes diagramas:

- Diagrama de Máquinas e Estados – apresenta os estados possíveis de um objecto, ocorridos em resposta a eventos;
- Diagrama de Sequências – apresenta as interacções que correspondem a um conjunto ordenado de mensagens trocadas entre objectos, descrevendo o seu comportamento e a forma como estes colaboram entre si;
- Diagrama de Comunicação – apresenta a organização estrutural das interacções dos objectos;
- Diagrama Temporal – apresenta a resposta da mudança de estado de um objecto a eventos externos, focando-se nas condições temporais;
- Diagrama de Visão Geral – apresenta de forma geral o fluxo de controlo dentro do sistema.

A estrutura de conceitos do UML é abrangente e fornece um vasto e variado conjunto de anotações que nos permitem definir:

- Elemento Básico - define os elementos, segundo a sua funcionalidade e a forma como estão organizados;
- Relação – define os relacionamentos entre os diferentes elementos, apresentando a sua semântica; e
- Diagrama – define as regras de agrupamento, segundo uma determinada lógica, dos elementos e respectivos relacionamentos.

2.2 Conceito de Refinamento

A noção de refinamento não é um conceito exclusivo do UML [3], vem dos primórdios de várias metodologias de desenvolvimento de software. Um dos primeiros mentores, foi nos anos 80 o Niklaus Wirth [26]. Segundo este os elementos básicos de refinamento

baseavam-se na d*ecomposição* na divisão de uma tarefa em subtarefas *e dados* em estruturas de dados.

Em cada etapa do refinamento está implícito uma série de tomadas de decisões, baseadas nos critérios de concepção do sistema.

Para representar a estrutura estática dum modelo, o UML fomenta os conceitos de perspectiva, abstracção e semântica, onde a junção da perspectiva com a abstracção permite uma descrição completa dum modelo, necessitando de um complemento de anotações que lhe permita tornar o seu nível semântico mais formal.

Nesta perspectiva, o nível de abstracção pode ser definido através da selecção das suas propriedades, dadas por classificadores que representam as características essenciais que distinguem as diferentes entidades que compõem um diagrama de classes.

Em [4], os modelos devem ser submetidos a um processo de comparação, de forma a se comprovar a sua compatibilidade sintáctica e semântica. Quando estamos perante alterações de um modelo, estas têm de ser consistentes. Isto possibilita que o aperfeiçoamento possa ser feito de forma gradual, mantendo-se coerente com o modelo original.

Nas diferentes etapas de desenvolvimento devem existir processos capazes de recriarem todo o percurso das actividades realizadas. Nesta óptica, a rastreabilidade assume um papel relevante, pois permite a reconstituição completa e exacta das várias etapas do desenvolvimento de um determinado modelo. No entanto, a rastreabilidade não garante que um modelo refinado seja consistente com o modelo original.

Assim, assegurando-nos da preservação semântica e do respeito das restrições impostas, podemos defender que os modelos resultantes de refinamento são consistentes. Por inferência, o seu desrespeito traduz-se na introdução de inconsistências dentro de um modelo.

Ainda relativamente à coerência, tomamos por premissa que um modelo coerente não tem de ser completo.

2.3 Técnicas do Refinamento

Nesta secção, analisamos algumas das técnicas de transformação existentes, que utilizam o conceito de refinamento.

Segundo a metodologia Catalysis [43], o refinamento entre classes pode ser realizado de duas formas diferentes: através dos atributos ou através das operações.

O refinamento das classes através dos atributos pode ser feito da seguinte forma:

- Adicionar novos atributos à classe refinada;
- Transformar um atributo numa nova classe podendo esta ter um conjunto de novos atributos;
- Transformar um ou mais atributo num atributo resultado refinado;
- Transformar um atributo num ou mais novos atributos.

O refinamento das classes através das operações ser feito da seguinte forma:

- Adicionar novas operações à classe refinada;
- Transformar uma operação numa ou mais operações.

2.3.1 Refactorização dos Diagramas de Classes

Muitas das organizações confrontam-se com a rápida desactualização dos seus sistemas e com a necessidade urgente de aperfeiçoamento e, por vezes, o crescimento dos mesmos. Isto porque, a evolução e respectivas adaptações dos sistemas são imprescindíveis a qualquer sistema actual.

A técnica de *refactorização* [6] baseia-se no processo de modificar um modelo, permitindo torná-lo mais legível ou conseguindo simplificar a sua estrutura, assegurando sempre, o seu comportamento e funcionalidades originais.

A refactorização está orientada para a depuração de sistemas, pois não tem por objectivo a correcção de erros, nem considera mecanismos para acrescentar novas funcionalidades, estando vocacionada para o melhoramento da qualidade do código.

As alterações feitas à estrutura interna são realizadas de forma gradual, permitindo a introdução de uma série de padrões de desenho, sem penalização no comportamento observado, reduzindo os custos, facilitando a reutilização e diminuindo o desperdício de certos recursos.

Em [6] é apresentado um conjunto de transformações de refactorização que preservam a semântica dos diagramas de classes UML, com notações OCL. Estas revelam-se úteis para a introdução de padrões num projecto. Para demonstrar a solidez das transformações, é definido uma tradução dos diagramas de classes e suas respectivas anotações para Alloy. São propostas um conjunto de regras baseadas na noção de equivalência que, quando aplicadas, definem uma transformação. Em [6], são propostas as seguintes regras:

- Introdução de generalizações;
- Introdução de associações com restrições;
- Promoção de uma classe a super-classe;

- Introdução de invariantes OCL.

Para obter um suporte formal, este trabalho utiliza o Alloy. A tradução apresentada centra-se no mapeamento de elementos dos diagramas de classes UML em construtores da linguagem de especificação formal Alloy, que serão analisados de forma automática, pelo Alloy Analyzer.

Ao nível operacional, as regras de tradução são divididas em duas fases distintas:

- Elementos gráficos do UML para os construtores Alloy;
- Invariantes do OCL para as fórmulas lógicas equivalentes em Alloy.

As regras propostas consideram as restrições de uma forma global, focando-se nos modelos de alto nível de abstracção, possibilitando assim, a comparação de modelos com os diferentes elementos. Isto é relevante porque torna possível a comparação da identidade de dois diagramas de classes, mesmo que estes tenham elementos diferentes.

2.3.2 Refactorização Automática

O interesse na utilização de refactorização automática tem vindo a aumentar, principalmente em projectos de grande escala. No entanto, devido à complexidade destes, esta contínua a mostrar-se insuficiente e limitada, tornando-se ineficaz porque as suas ferramentas ainda não fornecem programas que suportam refinamentos.

Isto obriga que, o utilizador necessite de conhecer profundamente toda arquitectura do sistema antes de proceder a um refinamento, o que na prática se traduz num consumo de recursos e tempo, para não incorrer no desrespeito das restrições do sistema, assim como no correcto funcionamento do sistema.

Em [7] é apresentado uma extensão de uma linguagem com descrições de refactorização. O sistema proposto é composto por um conjunto de ferramentas de suporte ao processo de descrições de refactorização, conseguindo produzir um ambiente de refactorização de acordo com estas descrições.

Esta linguagem de transformação possui um conjunto de operadores concebidos para agilizar as operações. Estes demonstram-se úteis no combate à complexidade inerente, comparação de padrões e reescrita dos termos de refactorização.

2.3.3 Precise UML

O Precise UML tem por objectivo desenvolver o UML como uma linguagem de modelação o mais precisa possível.

Os esforços deste projecto residem no desenvolvimento de novas teorias e práticas para clarificar e tornar mais precisa a semântica da UML; verificar a correcção dos projectos UML e construir instrumentos para apoiar a aplicação rigorosa do UML.

Nos últimos cinco anos, este grupo tem feito grandes esforços e promoção na formalização do UML.

Como referido, a falta de formalização do UML limita o suporte automático à análise. A proposta [8] apresenta uma técnica de análise rigorosa dos diagramas de classes UML. Ela é baseada na utilização de transformações dedutivas nos diagramas, onde são apresentadas regras e fornecidas as condições de verificação da solidez e da validade, das transformações realizadas.

Ao nível do UML, a notação gráfica não consegue providenciar uma interpretação precisa do seu significado, levando a várias interpretações e incorrecções no desenvolvimento e nos sistemas, com efeitos nefastos que prejudicam a análise e podem proliferar inconsistências.

Nesta óptica e com o intuito de colmatar a ambiguidade, a técnica descrita em [8] consegue aumentar a precisão semântica. Isto permite que um diagrama de classes UML seja representado como um modelo estático, onde a tradução origina um novo modelo com a propriedade deduzida ou com uma determinada conclusão. Neste âmbito é demonstrada e provada a solidez das suas regras.

Como já referido, um diagrama de classes representa uma perspectiva do sistema. O seu cerne é saber, sem ambiguidades, "o que as relações representam" para o sistema. Para atingir esse objectivo é necessário introduzir restrições nas relações entre as classes, condicionado a forma como os objectos interagem uns com os outros.

Assim, ao nível do significado das deduções devemos assumir o seguinte:

- Cada classe deve estar associada a um conjunto de instâncias;
- Cada instância deve ter uma representação única;
- As associações devem representar um conjunto de ligações entre as instâncias das classes.

A formalização do *significado* deve ter presente o conjunto de atribuições que satisfazem os elementos de um diagrama de classes. Para a manipulação das propriedades de um diagrama, foi apresentado o seguinte conjunto de regras de transformação:

- Substituição de uma classe;

- Substituição de uma associação;
- Promoção de uma associação; e,
- Remoção de uma associação.

Esta é uma técnica interessante na forma como aborda o potencial de dedução, no entanto, é limitada a um conjunto específico de elementos de um diagrama de classes. Neste caso, a preservação semântica não é garantida e não existe suporte automático de verificação da consistência entre modelos.

2.3.4 Evolução Consistente de Diagramas de Classes

Os modelos são excelentes artefactos que nos permitem, de uma forma independente, resolver lacunas estruturais dos sistemas. No entanto, quando existem vários modelos, nomeadamente diagramas de classes, existe a dificuldade de como manter esses modelos consistentes após várias etapas de refinamento.

Neste contexto, Alexander Egyed [9] propôs uma técnica de verificação automática da consistência de modelos após vários refinamentos. A sua abordagem divide a verificação da consistência em duas fases distintas: a fase da transformação e a fase da comparação.

A separação destas duas fases - *transformação* e *comparação* - permite tornar o processo mais transparente e flexível, pois desta forma facilita e simplifica as tomadas de decisão, tanto nos casos de reformar um modelo, como nos casos de resolução das inconsistências detectadas. Esta abordagem, também, permite a propagação das actualizações entre os modelos, de uma forma acessível e compreensível.

A fase da transformação baseia-se na tradução de elementos de um modelo com intuito de simplificar a sua comparação. Nesta óptica, utiliza a abstracção para suprimir os detalhes que não interessam. Neste trabalho, a abstracção é categorizada em dois tipos distintos: a composicional e a relacional. A abstracção composicional é idêntica à decomposição hierárquica dos sistemas, enquanto a abstracção relacional tem em foco as relações entre classes.

Na fase da comparação, parte-se do pressuposto que a transformação realizada foi bem sucedida, isto é, a transformação de um diagrama de baixo nível é "*algo semelhante*" ao diagrama de alto nível. Como a comparação é feita elemento a elemento, consegue-se facilmente detectar inconsistências.

Em [9] recorre a engenharia reversa e apresenta dois diagramas de classes onde o processo de *interpretação* é relativo ao diagrama de classes abstracto de baixo nível.

Este diagrama é o resultado das transformações de refinamento logo o modelo destino. O processo de *realização* é relativo ao diagrama de classes de nível superior, o que consideramos como modelo original.

Neste caso, antes de proceder ao processo de transformação tem de existir informação inicial de rastreabilidade que permita comparar a consistência do diagrama da *realização* com o diagrama da *interpretação*. A rastreabilidade serve de guia aos processos de transformação e de comparação.

O tratamento da quantidade de detalhe da informação recolhida, pode ser uma tarefa exaustiva e difícil, mas com a utilização de forma independente, destas duas técnicas, é possível utilizar a reutilização.

A rastreabilidade demonstra ser vantajosa ao nível de desempenho, principalmente quando a informação não é alterada, porque se beneficia de uma recolha prévia de informação obtida antes do processo de transformação. Uma outra vantagem é a utilização das regras de forma simples e genérica.

Ao nível da detecção das inconsistências, esta pode ser feita logo na etapa de transformação. Segundo o autor, os resultados obtidos são precisos, rigorosos e computacionalmente eficientes. No entanto, ainda existem lacunas a superar no suporte à propagação das alterações e à rastreabilidade.

2.3.5 Análise da Rastreabilidade em Modelos de Refinamento

O Software Quality Engineering Laboratory (SQUALL) é um centro de investigação orientado para pesquisa e desenvolvimento de novas metodologias e ferramentas de Engenharia de Software. Estas são concebidas para desenvolver software de alta qualidade a custos controlados, adoptando uma abordagem rigorosa.

Muitos dos seus projectos utilizam o UML como linguagem de modelação com o intuito de suportar uma vasta variedade de actividades de engenharia de software, e promovendo assim a abstracção. Tem a premissa que, se os modelos de especificação e desenho são providenciados de forma adequada, é possível alterá-los e testá-los ao longo do ciclo de desenvolvimento sem grandes custos.

Durante a fase de desenvolvimento de sistemas complexos, é necessário utilizar-se diferentes tipos de diagramas, interdependentes entre si, para modelar diferentes aspectos de um modelo. Neste âmbito, as alterações produzidas não devem afectar a consistência destes. Em [41] é analisado o impacto destas alterações ao nível da abstracção.

Ao nível da abstracção, a análise de impacto pode ter dois aspectos distintos: impacto horizontal e o impacto vertical. No nível do impacto horizontal, o foco reside nas alterações feitas ao mesmo nivel hierárquico. No nível do impacto vertical, o foco reside nas alterações realizadas a um nível e qual o seu impacto noutro nível. Nesta abordagem, é necessario existir a noção de rastrebilidade, de forma a estabelecer uma ligação entre os elementos do modelo nos diferentes niveis de abstracção. Por exemplo, o modelo de análise pertence a um nivel de abstracção superior relativamente ao modelo de desenho.

A rastreabilidade pode ser feita de forma manual ou automática. Na forma manual, e na prática a forma mais utilizada, a rastreabilidade é criada e da responsabilidade exclusiva da equipa de desenvolvimento. Na forma automática, a rastreabilidade é criada a partir da comparação entre dois modelos e das ligações estabelecidas entre eles.

Em [41] a rastreabilidade é automática e estabelecida entre dois modelos em diferentes níveis de abstracção.

O processo inicia-se através da identificação de transformações a realizar no modelo de análise e quais as consequências das transformações no modelo de desenho; de seguida, é analisado os refinamentos derivados destas e para finalizar o processo é estabelecida a rastreabilidade entre os elementos dos diferentes modelos.

2.3.6 OhCircus

O OhCircus [14] é uma linguagem de especificação formal, orientada a objectos, que reúne características da teoria Z, CSP [27] e cálculo de refinamento [26], com capacidade de representação de classes e heranças.

Em [14], os diagramas de classe UML são traduzidos em elementos construtores do OhCircus, sob as premissas de preservação semântica de todos elementos da estrutura dos diagramas (os relacionamentos e as invariantes globais). Nesta aproximação, é proposto a utilização de associações, numa visão abstracta dos diagramas de classes, em que, o processo de refinamento é feito através da eliminação das associações recorrendo à introdução de atributos nas classes.

No OhCircus, a informação é representada por uma sequência de esquemas capazes de definir processos, classes e métodos. Tem também qualificadores que providenciam propriedades aos esquemas.

Em OhCircus, a "perspectiva estática" é capturada directamente através do seu conjunto de classes. A interacção entre duas classes só pode ser captada através dos seus atributos, uma vez que não existe a noção de associação.

Uma outra restrição importante é a impossibilidade de estabelecer invariantes globais. As classes só podem restringir localmente os seus atributos.

Em [14], a solução proposta recorre à introdução de uma meta-classe denominada *Model*, responsável por conseguir capturar toda a estrutura do diagrama de classe: o conjunto de instâncias das classes, os seus relacionamentos, e os seus invariantes globais. Esta classe *Model* não faz parte do diagrama da classe, simplesmente faculta a sua interpretação. Assim, cada instância do modelo reflecte directamente um objecto do diagrama.

Ao nível do mapeamento, são disponibilizados dois tipos de esquemas: o esquema de estado e o esquema de operação. A tradução dos atributos de uma classe de UML são mapeados em *variáveis do esquema de estado* de uma classe de OhCircus, juntamente com a sua *visibilidade* e com o seu *tipo*. Os métodos são mapeados em esquemas de operação.

O OhCircus suporta explicitamente o conceito de herança.

O OhCircus recorre ao recurso de “representar os relacionamentos como atributos”, transformando os modelos de forma que estes consigam integrar os atributos e os relacionamentos. No entanto, o OhCircus não tem ferramenta de suporte usando o Z/EVES [14].

A principal contribuição de [14] é a transformação de diagramas de classe UML em especificações OhCircus com recurso de herança. A definição da meta-classe *Model* possibilita a captura de relações, invariantes globais e de determinados aspectos dinâmicos. Uma outra contribuição deste trabalho, reside na forma como é feita a análise do refinamento em UML. A meta-classe *Model* permite explorar o refinamento segundo a teoria Z.

2.4 Métodos Formais

No processo de desenvolvimento, os métodos formais podem ser eficientes e produtivos [10,11,14]. No entanto, a sua forte componente matemática, entre outros factores, são obstáculos que limitam e penalizam a sua utilização.

Uma das estratégias usada para atrair os analistas e tornar os métodos formais mais apelativos à indústria é sua junção com linguagens de modelação gráficas.

Os métodos formais destacam-se por serem técnicas matemáticas utilizadas para o desenvolvimento, especificação e verificação de sistemas. A utilização dos métodos formais suporta os seguintes aspectos [10]:

- Verificação ou especificação formal;
- Utilização da lógica matemática, promovendo a construção de sistemas compreensíveis e robustos;
- Aumento de poder de abstracção; e
- Validação através de simulações capazes de detectar inconsistências e ambiguidades.

Sendo inquestionáveis os benefícios da sua utilização, devemos reflectir, também, sobre as suas desvantagens:

- Suporte ineficiente de ferramentas;
- Pouca integração com outros métodos já existentes;
- Não possuem interfaces amigáveis e exigem treino e formação matemática;
- As provas formais requerem conhecimentos matemáticos sofisticados.

Mas, se uma especificação não for correcta ou precisa, pode-se incorrer na introdução de inconsistências e ambiguidades que produzem degradação no sistema e que podem ser nefastas.

A comunidade de métodos formais define as relações de refinamento da seguinte forma:

- enfraquecimento das pré-condições para aumentar a aplicabilidade das operações; e
- fortalecimento das pós-condições para diminuir o indeterminismo.

2.4.1 Método B

O método B [13] é uma linguagem de especificação formal, não orientada a objectos, baseada em *estados*, com fortes mecanismos que providenciam estruturação de especificações e com boas ferramentas de apoio ao desenvolvimento e à análise. O método B propõe o desenvolvimento de sistemas com base em refinamentos sucessivos, desde a especificação de um sistema até à sua implementação.

O modelo B providencia uma série de construtores que permitem a decomposição de uma especificação e, consequentemente, dos seus refinamentos. Estes construtores incluem, ao nível mais abstracto, uma máquina e vários níveis de refinamento, e, ao nível mais concreto, uma implementação.

Uma implementação pode utilizar vários módulos B, ao invocar as operações abstractas das suas máquinas. Um componente B permite que um estado abstracto possa ser dividido em várias partes, de forma que essas partes possam ser encapsuladas tornando mais fácil a sua compreensão e manipulação.

Em B, um invariante é uma propriedade do estado, que deve ser mantida pelas operações. Os invariantes também são utilizados para definir o tipo de cada variável.

Nas operações da máquina, as pré-condições definem o tipo dos argumentos, sendo possível utilizar "guardas" adicionais nos argumentos ou no estado das variáveis.

As operações definem o comportamento através de *substituições* que mostram como as variáveis do estado final da máquina dependem do seu estado inicial.

Ao nível prático, existem várias ferramentas de suporte, sendo a principal o Atelier-B. Esta ferramenta fornece suporte a cada etapa de desenvolvimento de um sistema. A verificação garante a correcção de todas as etapas de desenvolvimento.

Numa primeira abordagem, pode-nos parecer que o método B tem características similares ao UML, nomeadamente ao nível do encapsulamento das operações associadas com as variáveis de estado. No entanto, para que haja uma correspondência tem de ser feito um mapeamento capaz de traduzir as classes para as máquinas B, isto porque o UML é orientado a objectos e B não.

Por não ser orientado a objectos, método B encapsula o mecanismo de máquinas e permite que as variáveis sejam agrupadas com as operações em que actuam. É, também possível, instanciar diferentes instâncias de uma máquina, por meio do mecanismo de renomeação.

Na maioria dos trabalhos de tradução de UML para B, as classes são mapeadas em máquinas B, onde as restrições são feitas através da inclusão de máquinas adicionais. Isto dá origem a uma estrutura de máquinas complexa e difícil de verificar.

Ao nível do refinamento, este é feito através de etapas e da decomposição hierárquica. Após a especificação inicial informal dos requisitos, utiliza-se a abstracção para capturar as propriedades essenciais do sistema. As etapas podem ser de dois tipos distintos:

- A especificação pode ser refinada ao detalhar as estruturas de dados ou as operações que actuam sobre essas estruturas de dados;
- A especificação pode ser decomposta de forma a ligar os refinamentos procedentes a uma ou mais máquinas abstractas.

Assim, podemos deduzir que uma especificação abstracta que aplique um destes tipos de refinamento torna-se cada vez mais concreta e detalhada.

Num projecto típico, temos a utilização dos vários níveis de refinamento e decomposição para especificar todas as exigências do sistema. Em cada etapa do refinamento, são geradas provas que servem para validar o refinamento realizado. Estas garantem que toda a informação adicionada respeita a integridade dos níveis antecessores. No entanto, esta prova não é totalmente automática, necessitando de intervenção humana, o que requer uma utilização avultada de recursos e esforço.

2.4.2 Notação Z

A linguagem Z [11] baseia-se na teoria de conjuntos e na lógica de predicados de primeira ordem. Os conceitos da teoria de conjuntos são utilizados para representar o estado de um sistema e a lógica de predicados é usada para representar o seu comportamento.

Esta notação oferece um mecanismo de estruturação denominada por esquema. Os esquemas podem ser divididos em módulos, permitindo o encapsulamento e reutilização dos mesmos.

Na construção de uma especificação podemos descrever as variáveis que representam o estado, as mudanças de estado, as pré-condições e as pós-condições.

Relativamente aos refinamentos, esta linguagem permite a construção de uma especificação abstracta do sistema usando elementos matemáticos, e a respectiva modelação das suas propriedades, para se conseguir construir uma segunda especificação mais detalhada proveniente da primeira.

O Z/EVES [12] é uma ferramenta que consiste em traduzir especificações na linguagem Z para fórmulas em lógica de primeira ordem e providencia:

- Verificador de sintaxe;
- Esquema de expansão;
- Condições de cálculo e prova por teorema.

2.5 Especificação Formal

As linguagens de especificação formal permitem descrever de forma rigorosa e precisa o processo de desenvolvimento de um sistema, segundo o nível de detalhe que se deseje. A sua utilização permite eliminar imprecisões e ambiguidades que possam

surgir, assim como minimizar os erros nos requisitos ao descobrir inconsistências e requisitos subespecificados numa fase posterior ao desenvolvimento.

2.5.1 ALLOY

O Alloy é uma linguagem de especificação formal, orientada a objectos, do tipo declarativo, permitindo expressar as restrições estruturais complexas e os seus respectivos comportamentos. Pode ser aplicada a conjuntos e a relações, com base numa semântica simples para os objectos e suas respectivas associações. Ela proporciona um suporte para análise totalmente automática, sucinta e parcial.

Relativamente à sua sintaxe, esta foi criada com a contribuição de outras linguagens, entre elas o OCL. No entanto, denota-se uma influência acentuada da notação Z, direccionado para a criação de micro-modelos.

Um modelo Alloy consiste na declaração de assinaturas e fórmulas. As fórmulas permitem construir a base semântica de um modelo. Estas são compostas por: factos, predicados, funções e asserções [17].

O Alloy providencia uma ferramenta de suporte de análise automatizada, baseada na lógica de primeira ordem, denominada por Alloy Analyzer [2]. O Alloy Analyzer permite a análise das propriedades do sistema através da busca de instâncias do modelo, passíveis de verificar se algumas das propriedades do sistema são satisfeitas. Isto é feito através de uma tradução automática do modelo para uma expressão booleana, que será analisada pelo seu decisor SAT (Booleans Satisfiability Problem)[1], embutido nesta ferramenta. O utilizador só terá de especificar qual o domínio de alcance, do modelo. Se uma instância não estiver de acordo com a sua asserção, estando dentro do âmbito do seu domínio, assume-se que esta não é válida. No entanto, se não for encontrada qualquer instância, a asserção é considerada inválida para um âmbito maior.

2.5.2 UML2ALLOY

O Alloy é uma linguagem de modelação baseada na lógica de primeira ordem, inspirada no método Z, com suporte à análise automática. O objectivo da utilização de modelos não é a descrição total de um sistema, mas sim, modelar aspectos, definir restrições e verificar propriedades.

[1] SAT – Um demonstrador que aplica um procedimento sistemático de busca através de contra-exemplos para explorar o espaço de um determinado domínio, procurando atribuições que satisfaçam as fórmulas em questão.

O UML2Alloy é uma ferramenta que permite a tradução de diagramas de classes UML/OCL em modelos Alloy.

Esta ferramenta foi desenvolvida segundo a metodologia MDA (Model Driven Architecture), empregando o conceito de metamodelo.

Na Figura 2 apresentamos o processo de transformação de um diagrama de classes UML para modelos Alloy. Segundo o metamodelo de UML/OCL, os diagramas de classes UML/OCL são submetidos à ferramenta UML2Alloy a qual implementa um conjunto de regras de transformação que permitem produzir modelos Alloy.

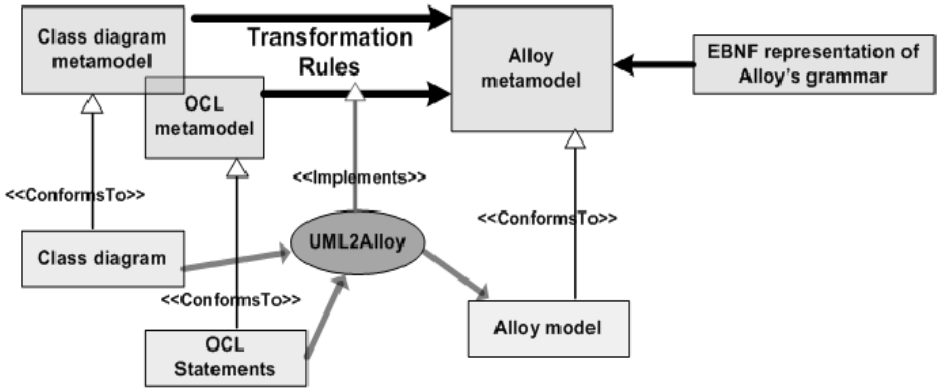


Figura 2: Ferramenta UML2Alloy

O UML2Alloy suporta as expressões OCL básicas, uma vez que as expressões em Alloy têm uma sintaxe semelhante. A expressão *if* tem uma pequena restrição: a cláusula *else* não pode retornar directamente o valor false. Este tem de ser o resultado de uma operação.

O UML2Alloy tem três ferramentas que servem de suporte à tradução de diagramas de classes em especificações Alloy. A primeira ferramenta permite editar modelos UML/OCL e exportar esses modelos para um ficheiro do tipo XMI. Este ficheiro é submetido à segunda ferramenta que o transforma num ficheiro de texto, servindo de dados de entrada à terceira ferramenta o analisador automático Alloy Analyzer.

A Figura 3 ilustra algumas regras de transformação adoptadas pelo UML2Alloy [35].

UML	Alloy
Classes	Signature Declarations
Attributes	Relations of the Signature
Data Types	Signature Declarations
OCL Expressions	Formula Expressions
If Expressions	If Formulas
Operations that return a type	Functions
Operations that return void type	Predicates
Operation Parameters	Parameters of Predicates or Functions
Associations	Relations of a Signature

Figura 3: Subconjunto de Regras de Transformação

Na Tabela 1, apresentamos o mapeamento de um elemento UML para um ou mais elementos em Alloy [35]:

Elemento UML/OCL	Elemento Alloy
Diagrama (Modelo)	Module
Classes	Assinatura
Atributo (Propriedade)	Assinatura – Campos
Generalização	Extends
Restrição OCL (Invariante)	Assinatura – Factos

Tabela 1: Mapeamento de UML/OCL para Alloy

O UML2Alloy consegue uma fusão das vantagens das duas linguagens. No entanto, ainda está em fase experimental.

2.6 Conclusão

Neste capítulo, apresentamos o estado de arte sobre linguagens de modelação e a existência de ferramentas para análise automática e suporte ao conceito de refinamento.

Para facilitar a comparação entre as diferentes linguagens, apresentamos a Tabela 2 como uma síntese das diferentes abordagens estudadas.

	Orientada Objectos		Mecanismos Representação	Suporte directo Refinamento		Ferramentas Análise	Tipos de Modelação
	Sim	Não		Sim	Não		
UML			Diagramas			Case - Rose	Visual
OCL			Restrições sobre UML			Atenas	Por Restrições
Precise UML			Diagramas			Não tem	Por Modelos
UML2ALLOY			Diagramas			Alloy Analyzer	Por Modelos
OhCircus			Classes			Indirecto -Z/EVES	Por Modelos
Z			Esquemas			Z/EVES	Por Modelos
Metodo-B			Estados			Atelier-B, B-Tool	Por Modelos

Tabela 2: Síntese das diferentes linguagens estudadas

As linguagens de modelação dividem-se entre as orientadas a objectos e as não orientadas a objectos, e as que providenciam, ou não, suporte directo ao refinamento.

A linguagem UML que não tem suporte de verificação formal. Assim, fomos direccionados para a selecção da uma linguagem de especificação formal orientada a objectos e que, preferencialmente, fizesse a análise e verificação formal dos modelos de forma automática. Nesta perspectiva, escolhemos a linguagem de modelação Alloy. Escolhemos o Alloy, tendo em conta o suporte para análise automática, podendo capturar e analisar de forma precisa e robusta toda especificação lógica de um sistema.

3 UML

Como referido, o UML é uma linguagem de modulação visual, apelativa e intuitiva, que providencia vários tipos de diagramas, com o intuito de representar todo a globalidade de um sistema de informação.

Na versão UML 2 [39], existem vários tipos de diagramas que nos permitem ter as três principais visões: a visão estrutural, a visão funcional e a visão dinâmica. No âmbito desta tese, centramo-nos nos conceitos da visão estrutural através do diagrama de classes, no entanto, para a descrição algorítmica utilizaremos o diagrama de actividades apresentado na secção 2.3, dando uma visão funcional do mesmo.

Ao nível estrutural, os diagramas de classes representam a visão geral de um sistema, através das entidades envolvidas e seus respectivos relacionamentos.

Na Engenharia de Software, o refinamento é expresso como um aperfeiçoamento, onde o componente aperfeiçoado substituirá o seu antecessor, sem modificar as propriedades do sistema. Neste processo, temos adição de informações, que no seu melhor, consegue que o modelo destino seja o mais próximo possível do modelo de implementação.

Os diagramas de actividades permitem especificar a dinâmica de um sistema. Estes representam uma serie de acções cujas transições são desencadeadas quando as acções terminam. Este diagrama é adequado para representar o comportamento algorítmico.

Na Engenharia de Software [15], ao nível do desenvolvimento, considera-se um processo ‘o *caminho para chegar a um fim*’ como uma sequência de actividades, agrupadas em fases e tarefas, executadas de forma sistemática e uniformizada, onde a partir de um conjunto de entradas produz-se um conjunto de saídas.

Considera-se que uma metodologia engloba o conceito de processo e recomenda um conjunto de procedimentos, ferramentas, técnicas e notações [45].

O UML não se enquadra na categoria de metodologia de desenvolvimento. Isto significa que o UML não fornece uma orientação formal de como devemos começar e quais devem ser os passos para conceber um sistema. No entanto, com a sua aplicação consegue-se aumentar a produtividade, maximizar a compatibilidade dos sistemas, simplificar o processo de desenvolvimento, promover a comunicação e facultar a informação produzida aos diferentes membros de uma equipa de trabalho.

Isto porque, providencia métodos que aumentam o nível de abstracção, promovendo que os modelos sejam simples e com incidência no domínio do problema. Na combinação do domínio do problema com uma descrição semântica formal, podemos aspirar à obtenção de um bom nível de automatização.

Como já referido, um modelo é uma simplificação da realidade [4,17,22], logo este, com os correctos mecanismos, pode descrever uma solução para qualquer tipo de sistema. Os diagramas são visualizações gráficas destes.

Na fase de análise do desenvolvimento de software, os modelos são os artefactos que expressam todo contexto de um sistema. Estes devem ser construídos segundo um determinado nível de detalhe, que poderá e deverá ser passível a refinamentos até que o sistema consiga atingir o grau de qualidade solicitada, logo aumentar o grau de confiança no software.

3.1 Modelação da Estrutura

No paradigma orientado a objectos, a modelação da estrutura de um sistema de informação consiste na identificação das classes e suas respectivas relações [39]. Para a especificação da estrutura de um sistema, o UML providencia os seguintes artefactos: as classes, as relações, os objectos e as interfaces. No âmbito deste trabalho, focaremos somente no diagrama de classes.

3.1.1 Diagrama de Classes

Segundo a abordagem orientada a objectos, os diagramas de classes são os artefactos responsáveis por apresentar a modelação de toda a estrutura estática dum sistema. Estes diagramas descrevem a estrutura interna das classes, assim como são estabelecidas as relações existentes entre as diferentes classes.

Classes

As classes são descrições de um conjunto de objectos que constituem um diagrama de classes. Esta descreve as regras que orientam o comportamento dos seus objectos.

Um objecto é uma instância de uma classe. Este herda as propriedades estruturais e comportamentos da classe, possuindo um estado em contexto de execução. O estado é dado pelos valores assumidos segundo o conjunto das propriedades estruturais da classe. O estado é dinâmico e varia ao longo do tempo, consoante o objecto interage com os outros objectos.

Ao nível de estrutura de modelação de uma classe, esta deve ter as seguintes propriedades internas: um nome; se é ou não abstracta; qual a sua visibilidade; quais os seus atributos; e quais os seus métodos.

Atributos

Um atributo representa uma propriedade intrínseca que todos os objectos, de uma classe, têm de ter. Cada objecto pode atribuir valores específicos aos seus atributos.

Ao nível de estrutura de modelação de um atributo, este deve ter as seguintes propriedades internas: um nome; qual o seu tipo; e qual a sua visibilidade.

Métodos

Na sua essência, os métodos revelam o comportamento da classe através da manipulação de dados de um objecto e da forma como estes executam as suas tarefas.

Um método tem por objectivo implementar uma acção que permite especificar o comportamento que uma classe pode adoptar.

Ao nível de estrutura de modelação de um método, este deve ter as seguintes propriedades internas: um nome; se é ou não abstracto; qual a sua visibilidade; o número de parâmetros; e qual o tipo do resultado.

Quando temos uma classe do tipo abstracto, esta não pode ser directamente instanciada, simplesmente pode fornecer características e funcionalidades que serão herdadas pelas classes que a estendem. Uma classe abstracta pode conter atributos e métodos. Os métodos podem ser concretos ou abstractos, dependendo se estão ou não implementados na classe abstracta. Na Figura 4 apresentamos um exemplo de uma classe.

Figura 4: Exemplo de uma classe

A propriedade de visibilidade indica como é feito o acesso a um elemento. Esta indica a forma como o elemento pode ser visualizado por classes externas. A visibilidade pode ser dos seguintes tipos:

- Privado - o elemento não pode ser acedido por nenhuma classe que não seja a própria,
- Protegido - o elemento só pode ser acedido por subclasses pertencentes à classe em questão, e
- Público – o elemento pode ser acedido sem restrições por outras classes.

Relações

Em UML, as relações servem para descrever a interligação entre as classes. Entre as classes, existem os seguintes tipos de relações: associação, agregação, composição, dependência e generalização.

Ao nível de estrutura de modelação de uma relação, esta deve ter as seguintes propriedades internas: um tipo e um nome.

Associação

Uma associação é um relacionamento estrutural entre objectos de classes diferentes, cuja informação tem de ser preservada. Esta especifica como os objectos de uma classe estão ligados aos objectos da outra classe, sem que haja a necessidade de um dos objectos ser uma parte do outro. Desta forma, a associação consegue descrever qual a semântica utilizada para a ligação dos diferentes objectos.

O mecanismo de associação traduz simplesmente uma ligação entre classes e o número de instâncias que interagem nessa ligação. A multiplicidade identifica o número de objectos que podem estar associados a cada vértice da associação.

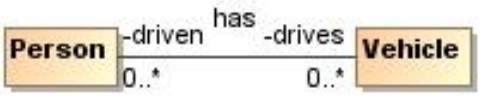


Figura 5: Relação de associação bidireccional entre classes

A associação pode-se dividir em dois tipos: composições e agregações.

Agregação

Uma agregação é um caso particular de uma associação. Uma agregação indica que uma das classes do relacionamento é considerada como um "*todo"* enquanto a outra classe é uma "*parte"* desse "*todo"*. Usualmente, identifica-se este tipo de associação atribuindo

às associações nomes identificativos, tais como "*é parte de um todo*", "*consiste em* " ou "t*odo contém parte de*".

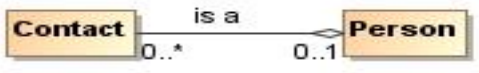


Figura 6: Relação de agregação entre classes

Composição

A composição é um outro caso particular de uma associação. A composição correspondem à noção de objectos agregadores, isto é, é do tipo agregação mas com uma relação mais forte. A classe agregadora é considerada o "*todo*" e as classes agregadas são consideradas como as suas "*partes*". Uma classe agregada só pode pertencer a uma classe agregadora e esta tem a capacidade de as criar e destruir. Qualquer alteração na classe agregadora irá afectar directamente todas as suas agregadas.

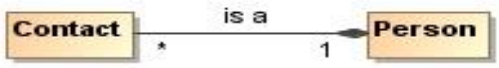


Figura 7: Relação de composição entre classes

Dependência

A relação de dependência indica-nos a ocorrência de uma modificação na especificação de uma classe, que afectará as restantes classes que interagem directamente com ela. Neste âmbito, um refinamento ao nível de código pode ser considerado como uma especialização de uma dependência.

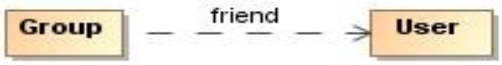


Figura 8: Relação de dependência entre classes

Generalização

A generalização é uma relação de herança permitindo a criação de classes especializadas. Esta relação refere-se a classes que partilham a mesma estrutura e o mesmo comportamento. Neste âmbito, uma generalização é um relacionamento entre uma classe geral (superclasse) e uma ou mais classes específicas (subclasse). Uma subclasse é mais específica que a sua superclasse, possuindo todas as características da sua superclasse, assim como podendo conter especificidades próprias, não partilhadas pelas restantes subclasses.

O mecanismo de generalização é definido em termos de "é um de". Nele configura a inclusão de classes que herdam as características e comportamentos da classe geral.

Figura 9: Relação de generalização entre classes

3.2 OCL

As restrições são artefactos essenciais com capacidade de providenciar algum formalismo à notação UML. No caso do UML, este utiliza o OCL [16], para conseguir especificar as restrições.

Uma restrição consiste na especificação de uma limitação a um elemento. Ela permite alterar a semântica assumida por omissão. O desrespeito destas condições impostas leva certamente a uma maior imprecisão, o que poderá originar inconsistências, incorrecções e incompletude nos modelos. Para descrever uma restrição, em UML, temos os seguintes artefactos: invariantes; pré-condições; e pós-condições.

Para modelar um sistema é necessário que se tenha um sólido conhecimento do problema a ser resolvido pois, só dessa forma se consegue determinar quais as propriedades relevantes do problema e como estas devem ser tratadas pelo sistema para que seja possível alcançar o objectivo desejado. Nesta óptica, assume-se necessária a sua verificação e respectiva validação.

O OCL [15] é uma linguagem de especificação formal de restrições, baseada na teoria dos conjuntos e na lógica de predicados, utilizada para descrever restrições em modelos UML.

O OCL é uma linguagem formal e precisa com o intuito de representar restrições semânticas. A sua estrutura está directamente ligada aos modelos UML. É composta por expressões que são do tipo declarativo no sentido que expressam "o quê" e não o "como" a expressão deve ser implementada. As expressões são escritas em forma de afirmações.

A avaliação de uma expressão resulta sempre num valor booleano, e este não tem capacidade de mudar o estado do sistema no modelo.

Tipicamente, estas expressões especificam condições invariantes de um sistema, não podendo alterar o estado do mesmo. Quando uma expressão OCL é avaliada, esta retorna um valor, o qual não interfere com o estado original do modelo.

O OCL providencia as seguintes especificações:

- invariantes (propriedades que caracterizam uma classe como um todo, sendo sempre verdadeiras);
- pré-condições (condições que devem ser satisfeitas para que uma dada operação seja executada com sucesso); e,
- pós-condições (condições garantidas por uma operação após o término de sua execução bem-sucedida).

As notações têm funcionalidades variadas, nomeadamente:

- servir apenas como documentação,
- definir restrições e propriedades a serem verificadas, e
- servir como modelo para um programa.

Um elemento penalizador é o facto de o OCL não ser de fácil aprendizagem e compreensão, principalmente na interpretação de expressões complexas. No entanto, para expressões simples e curtas, consegue proporcionar um formalismo satisfatório.

3.3 Modelação do Comportamento

Segundo o paradigma orientado a objectos, a modelação do comportamento de um sistema de informação consiste na modelação inter-objectos (identificação dos padrões de troca de mensagens) e na modelação intra-objectos (identificação de eventos em que um objecto possa encontrar durante o seu ciclo de vida [39]). A modelação do comportamento é a mais adequadas para a descrição algoritmos de implementação.

3.3.1 Diagrama de Actividades

Para a modelação do algoritmo a implementar o sistema de regras de refinamento, utilizaremos o diagrama de actividades (modelação de comportamento inter-objecto). A escolha deste diagrama baseia-se na necessidade de especificar o comportamento do nosso sistema de informação como um todo, utilizando os diferentes tipos de mecanismos de abstracção que este tipo de diagrama oferece.

Segundo alguns especialistas, o diagrama de actividades é o mais adequado à modelação da visão funcional de um sistema. Isto porque permite a descrição lógica dos seus processos ou funções. Assim, o diagrama de actividades permite descrever o comportamento especificando a sequência de operações e decisões, assim como determinando quando estas devem ser utilizadas.

Em termos gerais, o diagrama de actividades consiste numa série de actividades ligadas por transições. Uma actividade é definida como um passo de um processo onde pode ser realizada uma determinada tarefa (calculo, manipulação de dados, pesquisa de informação, envio/recepção de dados). Uma transição representa o término dessa actividade e indica qual a actividade seguinte ou o término do processo.

Num diagrama de actividades, o fluxo é conduzido para a realização e conclusão de uma operação. Numa versão mais simplificada, este tem correspondência com os fluxogramas. No entanto, o diagrama de actividades permite um grau de especificação diferente, mais abrangente e representativo, detalhando a informação consoante as necessidades do analista (uma visão generalista ou uma visão mais detalhada).

Ao nível funcional, os diagramas de actividades providenciam as seguintes funcionalidades:

- Acções;
- Actividades não atómicas;
- Transições;
- Objectos;
- Nós de decisão, difusão e junção.

Na Figura 10 apresentamos um exemplo genérico de um diagrama de actividades.

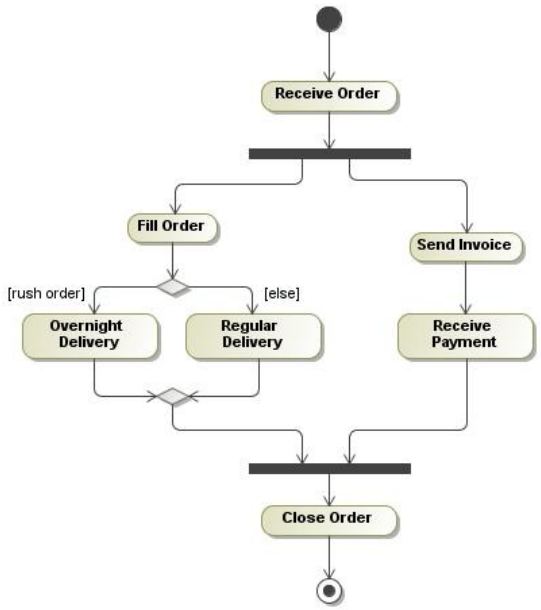


Figura 10: Exemplo de um diagrama de actividades [28]

Nos diagrama de actividades é possível utilizar um conjunto de mecanismos de permitem modelar as nossas especificações, são eles: a tomada de decisão, a

concorrência, partição das diferentes actividades e interacção entre os diferentes objectos.

O mecanismo de tomada de decisão consiste em especificar se uma determinada actividade deve ser realizada após a execução da actividade corrente. Esta especificação suporta uma condição de guarda, colocada de forma adjacente à transição do estado correspondente.

O mecanismo de concorrência consiste em especificar um conjunto de actividades que tem de ser obrigatoriamente realizadas, no entanto não existe qualquer tipo de ordenação temporal para a sua execução. Este mecanismo suporta dois conceitos distintos: a difusão e a junção de actividades.

3.4 Metamodelo do Diagrama de Classes UML

Um metamodelo é uma estrutura que nos permitam definir um conjunto de elementos construtores de uma linguagem. A transformação (mapeamento) é uma ferramenta que permite a manipulação de um modelo, entre diferentes linguagens.

Através dos metamodelos conseguimos atingir a precisão e compreensão da semântica da modelação dos diferentes elementos. Um metamodelo permite descrever a estrutura interna dos modelos, criando a sua sintaxe e explicando a semântica das suas entidades e das suas relações [34].

Recorrendo à utilização de metamodelos, em linguagens não formais, conseguimos eliminar a ambiguidade e reforçar a capacidade de formalização da sua semântica.

No âmbito dos diagramas de classes UML, o metamodelo proposto deve expressar o molde de como as classes, os atributos, as operações e os relacionamentos devem ser traduzidos para os modelos Alloy. Neste contexto, é crucial que o metamodelo proposto consiga capturar todos os aspectos estáticos de um sistema.

O MOF é um standard do MDE (Model Driven Engineering) da OMG, que permite a criação de metamodelos, fornecendo a sintaxe abstracta e semântica, isto é um método que permite definir a estrutura da linguagem ou dos dados. É projectado numa arquitectura de metamodelação composta por quatro camadas de abstracção. A primeira camada (camada superior) fornece um meta-metamodelo, a segunda camada fornece uma linguagem para a criação dos metamodelos, a terceira camada descreve os elementos dos modelos segundo os metamodelos criados na camada anterior, e a última camada descreve os objectos. É através deste tipo de arquitectura que é possível a correspondência de cada elemento de um modelo, em cada camada.

Para implementar a transformação dos diagramas de classes em modelos Alloy utilizamos a arquitectura MOF para a criação do metamodelo do modelo destino.

3.5 Conclusão

Neste capítulo, apresentamos a linguagem de modelação gráfica UML e sua relevância na Engenharia de Software.

Embora o UML não seja uma metodologia por não fornecer um conjunto formal de procedimentos iterativos, tem-se demonstrado muito útil no aumento de produtividade, na maximização da compatibilidade entre os sistemas, na simplificação do processo de desenvolvimento, na promoção da comunicação e da informação produzida pelos diferentes membros de uma equipa de trabalho.

Na modelação da estrutura, apresentamos o diagrama de classes e os respectivos conceitos básicos como o tipo de diagrama capaz de descrever as entidades e suas relações que constituem um sistema. Sendo o UML uma linguagem de modelação semi-formal, a linguagem de restrições OCL permite a representação das restrições semânticas no UML.

Na modelação de comportamento, apresentamos o diagrama de actividades como o tipo de diagrama capaz de descrever o algoritmo de implementação do sistema de regras de refinamento proposto.

Finalizamos este capítulo, com uma descrição do conceito de metamodelo.

4 Alloy

Neste capítulo descrevemos e analisamos a linguagem Alloy e a sua ferramenta associada, Alloy Analyzer.

Na secção 4.1 apresentamos os princípios básicos da linguagem de modelação Alloy; na secção 4.2 apresentamos a sua lógica; na secção 4.3 apresentamos a sua modelação; secção 4.4 analisamos a sua ferramenta de suporte de análise automática, Alloy Analyzer; na secção 4.5, fazemos uma comparação entre a linguagem UML e a linguagem Alloy e finalizamos com uma conclusão dos tópicos abordados.

4.1 Princípios Básicos

Como referido, o Alloy é uma linguagem de especificação formal, orientada a objectos, alicerçada na lógica relacional, com uma notação baseada em conceitos simples e precisos, com capacidade de análise automática.

Esta linguagem integra-se na categoria dos métodos formais, podendo capturar e analisar de forma precisa e robusta toda especificação lógica de um sistema. Nesta abordagem, um sistema pode ser descrito através de um conjunto de restrições estruturais e respectivos comportamentos [29].

Os modelos Alloy são concisos, analisáveis, declarativos e estruturais. A principal mais-valia dos modelos é serem simultaneamente declarativos e analisáveis.

Um modelo declarativo [29] descreve um estado de um sistema, listando as suas propriedades e restrições. No entanto, não específica como os estados são construídos ou como se transita de um estado para outro.

Relativamente à sua semântica, um modelo pode ter naturezas diferentes: lógica, estática ou dinâmica. Assim, a verificação da consistência de um modelo deve feita segundo a sua natureza.

A linguagem Alloy é aplicada a conjuntos e a relações, com base numa semântica simples para os objectos e suas respectivas associações. Assim, todas as estruturas são construídas a partir de um conjunto de átomos. Os átomos são indivisíveis, imutáveis e elementares, logo assumem-se como entidades atómicas.

As relações são as estruturas que relacionam os átomos. Matematicamente, são um conjunto de pares, onde cada par consiste em dois ou mais átomos, numa ordem específica. Ao nível do Alloy, as relações consistem num conjunto de túplos, onde cada túplo é uma sequência de átomos.

Na Tabela 3 apresentamos um modelo Alloy cujo objectivo é explorar uma agenda de contactos. Cada entrada da agenda mapeia um nome no seu endereço.

Modelo Alloy	*Descrição*

`module tour/addressBook`
 	
`sig Name, Addr { }`
`sig Book {`
`  addr: Name -> lone Addr`
`}`
 	
`fun lookup(b: Book,n:Name):set Addr{`
`  n.^(b.addr) & Addr`
`}`
 	
`assert lookupYields {`
`  all b: Book, n: b.names |`
`  some lookup(b,n)`
`}`
 	
`check lookupYields for 4 but 1 Book`
	Temos as assinaturas: *Name, Addr, Book*, cada uma representa um novo tipo e um novo conjunto de objectos.
	A assinatura *Book* tem a relação *addr*. A *addr* é uma relação do tipo *Book* que mapeia os nomes do tipo *Name* para um endereço do tipo *Addr*.
	Temos a função *lookup* que tem como dados de entrada a agenda e o nome e como dado de saída um conjunto de endereços.
	A asserção *lookupYields* pesquisa todos os nomes na agenda.
	O comando *check* que permite verificar se a asserção é válida num determinado âmbito de pesquisa, neste caso temos 4 objectos para a assinatura *Book*.

Tabela 3: Exemplo de um modelo Alloy

Como referido, em UML, as relações definem uma ligação entre classes, em Alloy uma relação é representada por um conjunto de túplos. Esta diferença deriva das diferentes filosofias adoptadas pelas respectivas linguagens: UML é vocacionada para representar conceitos de programação orientada a objectos; o Alloy é direccionado para as restrições e análise abstracta de especificações.

4.2 Lógica e Linguagem

O seu mecanismo de lógica permite-nos expressar as abstracções de forma flexível. Existem dois tipos de abstracções: as restrições e as expressões.

As restrições são limitações, implícitamente sempre verdadeiras, impostas aos átomos. As expressões são os conjuntos obtidos das relações.

A este nível, Alloy oferece três estilos distintos de aplicação com a mais-valia de puderem ser combinados entre si: o *cálculo de predicado, as expressões de navegação e o cálculo relacional*.

O estilo de *cálculo de predicado* permite dois tipos de expressões:

- os nomes das relações que são usadas dentro dos predicados; e
- os túplos formados a partir de variáveis quantificadoras.

O exemplo seguinte ilustra uma restrição que indica que, se dois objectos Book têm o mesmo endereço então estes objectos são iguais.

Ex: **all** b1, b2: Book | b1.addr = b2.addr **implies** b1=b2

As *expressões de navegação* representam conjuntos formados por variáveis quantificadas que permitem "navegar" através das relações.

Ex: **all** b: Book | **lone** b.addr

No *cálculo relacional*, as expressões representam relações, sem qualquer tipo de variáveis quantificadoras.

Ex: **no** ~(n.^(b.addr)).(n.^(b.addr)) - **iden**

A lógica relacional de Alloy, permite combinar os quantificadores da lógica de primeira ordem com os operadores relacionais de cálculo.

4.3 Modelação Alloy

Um modelo representado em Alloy corresponde a um conjunto de instâncias válidas. Uma instância é uma ligação de valores a variáveis. Normalmente, uma única instância representa um estado ou vários estados [37]. Os modelos são estruturados segundo os seguintes elementos: as assinaturas e as fórmulas [36].

4.3.1 Assinaturas

Uma assinatura representa um conjunto de átomos, com um nome único e um corpo. No seu corpo, podem ser declarados tipos, relações e restrições.

As assinaturas têm campos e cada um deles representa uma relação. É através da declaração de assinaturas que podemos criar uma hierarquia de conjuntos. No entanto, existe uma limitação ao nível de heranças, uma vez que o Alloy não suporta herança múltipla.

4.3.2 Fórmulas

As fórmulas são expressões que possibilitam a declaração do "conhecimento" de um sistema, em Alloy, i.e., a base semântica de um modelo. As fórmulas podem ser dos seguintes tipos: factos, predicados, funções ou asserções.

- Um facto representa restrições semânticas que são sempre aplicadas. Logo, estes quando existem são automaticamente assumidos. Para a sua definição utiliza-se lógica de primeira ordem.
- As funções e os predicados têm objectivos idênticos e são análogos aos métodos de uma classe. Ambos podem ser utilizados para especificar o comportamento, assim como podem ser aplicados para ligar factos a expressões. Através destes, Alloy permite-nos capturar as restrições dos átomos que devem ser satisfeitas antes e depois da sua execução. No entanto, as funções distinguem-se dos predicados pelo tipo de resposta que retornam. Os predicados retornam valores booleanos, enquanto as funções retornam valores segundo um determinado contradomínio.
- Uma asserção é uma expressão booleana sempre verdadeira. Esta tem por objectivo verificar se as condições ou as propriedades, baseadas nos factos, são válidas para toda e qualquer instância do modelo. Neste âmbito, é necessário reforçar determinadas restrições quando não são directamente expressas, mas que tem de ser verificadas. Quando uma asserção retorna um valor verdadeiro significa que a condição é válida para qualquer instância. Caso contrário, significa que a restrição é inválida e é gerado um contra-exemplo.

  No caso do contra-exemplo, o cenário produzido pode indicar que os factos são suportados, mas não a asserção. No entanto, é possível que não haja nenhuma falha no domínio proposto, mas asserção poderá falhar quando é aumentado o espaço do domínio.
- O comando *run* específica o âmbito de busca para as instâncias, fornecendo as simulações válidas obtidas dos predicados. Por exemplo: *run* for 3 significa que para cada assinatura pode-se ter até 3 objectos diferentes. Com "*exactly" numero* limitamos o numero de objectos por assinatura.
- O comando *check* verifica a validade de uma asserção, tendo um comportamento e funcionamento idêntico ao *run*. Este é somente usado nas asserções, gerando um contra-exemplo quando esta não é válida.

Na Figura 12 apresentamos o metamodelo de Alloy [35].

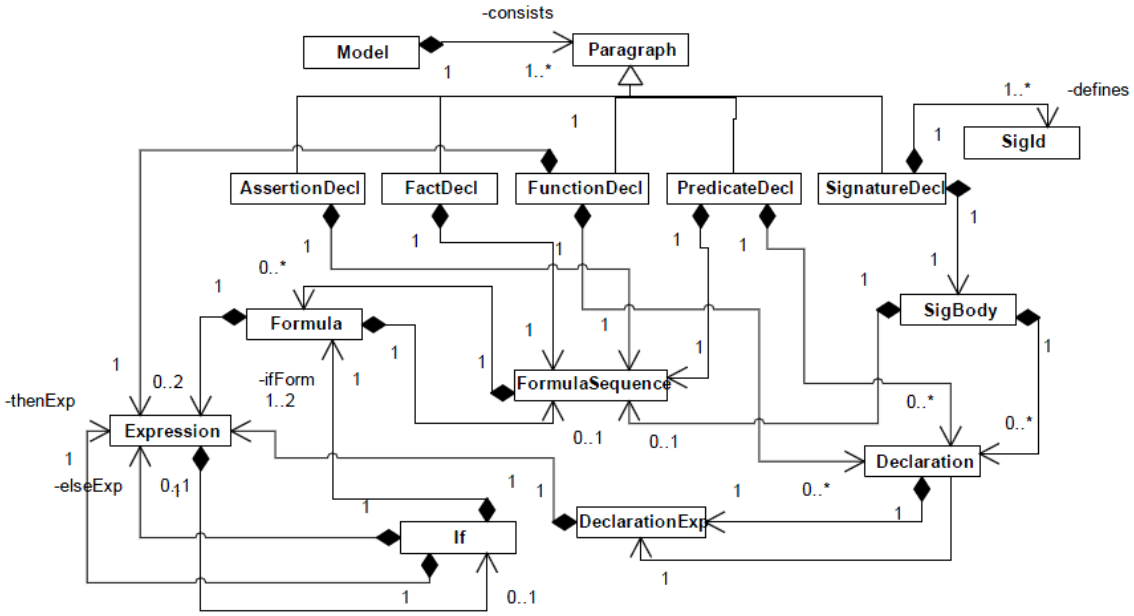


Figura 11: Metamodelo Alloy.

Em Alloy um *Model* consiste num ou mais *Paragraph*. Existem vários tipos de *Paragraph*: *SignatureDecl, PredicateDecl, FunctionDecl, FactDecl e AssertionDecl*. A SignatureDecl é idêntica à Classe do UML, declara uma assinatura que denota um conjunto de átomos. Tem um identificador único, *SigID*, e tem um corpo, *SigBody*, que define as declarações, *Declaration*, e as expressões, *DeclarationExp*, que consiste numa ou mais expressões. Um PredicateDecl, pode consistir numa *Declaration* ou numa *FormulaSequence*. A *FormulaSequence* consiste em *Formula* que, por sua vez, consiste numa ou mais expressões. A *FunctionDecl* consite numa Declaration, a FactDecl é uma expressão sempre válida, e *AssertionDecl* é uma expressão Booleana.

4.4 Alloy Analyzer

O Alloy Analyzer [37] é um analisador de restrições de suporte à linguagem Alloy. Este permite realizar dois tipos de análise sobre os modelos:

(i) Verificar as instâncias do modelo;

(ii) Verificar as asserções.

Esta ferramenta compila e analisa os modelos escritos em Alloy, baseando-se na lógica de primeira ordem e algoritmos SAT que permitem a verificação e validação de modelos formais.

O Alloy Analyzer permite a análise das propriedades do sistema através da busca de instâncias do modelo. Esta análise é realizada através de uma tradução automática do modelo para uma expressão booleana, que será analisada pelo seu decisor SAT,

embutido nesta ferramenta. O utilizador só terá de especificar qual o domínio do modelo.

Se uma asserção produzir um resultado falso dentro do domínio, assume-se que esta não é válida. No entanto, se produzir um resultado verdadeiro, a asserção pode ser inválida quando aumentamos o domínio.

Contrariamente ao método de prova por demonstração por teorema, o Alloy Analyzer realiza uma análise automática que não é completa nem exaustiva porque a pesquisa simplesmente examina um espaço limitado de casos. Este analisador usa a tecnologia de resolução de restrições, conseguindo produzir instâncias com uma vasta cobertura. Neste âmbito, o utilizador só precisa de providenciar a propriedade que pretende que seja verificada, e o analisador gera simulações de forma a encontrar exemplos de execuções que a satisfaçam ou contra-exemplos, quando uma instância viola a propriedade em análise.

A pesquisa de instâncias é conduzida segundo as dimensões especificados pelo utilizador, atribuindo um vínculo ao número de objectos de cada tipo. Ao nível prático, tem-se verificado que uma pesquisa com um âmbito de pequena dimensão consegue obter um vasto âmbito de experimentação que, na maioria das vezes, é o suficiente para se encontrar erros imperceptíveis.

O Alloy Analyzer analisa as restrições através de associação da uma restrição descrita num predicado com os factos declarados explicitamente nos parágrafos ou dos factos implícitos nas declarações.

O Alloy Analyzer permite a simulação e verificação de modelos. Ao nível da simulação, esta é realizada através de uma declaração de lógica de primeira ordem, correspondendo ao comando *run*. Através deste comando o analisador gera uma instância em conformidade com o modelo proposto. Numa instância de um modelo temos a simulação de um cenário onde os predicados e factos são válidos.

A simulação pode ser usada para demonstrar a consistência de um determinado predicado, gerando uma instância do modelo. Neste contexto, o utilizador limita o âmbito dos objectos que uma instância do modelo deve ter. A ausência de um exemplo implica que provavelmente o modelo é inconsistente .

Numa simulação o alcance do âmbito é um factor relevante. Desta forma limita-se as dimensões das assinaturas, assim como da pesquisa exaustiva de exemplos ou contra-exemplos.

Ao trabalhar-se com a instanciação (declaração implícita dos factos) consegue-se obter uma cobertura mais razoável que as técnicas tradicionais técnicas de testing. A instanciação é eficaz na descoberta de erros subtis, principalmente pela apresentação dos contra-exemplos produzidos.

A verificação é utilizada para provar a legitimidade das asserções. Como estas têm de ser sempre verdadeiras, quando o analisador gera um contra-exemplo significa que a instância produzida é inválida, logo a asserção é falsa. As asserções servem para verificar se o modelo satisfaz ou não uma determinada propriedade do sistema. Quando o Alloy Analyzer verifica as asserções, analisa todas as possíveis instâncias do modelo. Ao nível da pesquisa, esta é obtida através da construção de uma fórmula booleana. Esta é completa para o âmbito do modelo e convertida na forma normal conjuntiva (CNF), cujo seu resultado é processado pelo SAT. Este procura encontrar uma atribuição de valores para as variáveis Booleanas que satisfaçam a CNF.

4.5 Comparação entre UML e Alloy

Tanto o UML como o Alloy são linguagens de modelação capazes de especificar requisitos de sistemas de software complexos.

Ao nível sintáctico, o Alloy é compatível com UML. A expressividade e complexidade do UML torna-o ambíguo, porque proporciona várias interpretações sobre o mesmo modelo, por sua vez, o Alloy é preciso, abstracto e conciso [31].

Sendo estas linguagens compatíveis uma com a outra, é possível coligá-las, tornando-as mais eficientes e eficazes, isto é, traduzir os diagramas de classes UML em modelos formais Alloy, permitindo que a análise e verificação possa ser feita automaticamente.

A análise e verificação automática deve ser realizada na fase de análise ou na de desenho do desenvolvimento, permitindo assim que a detecção de erros seja feita atempadamente, de forma a facilitar a sua respectiva correcção e para que estes não repercutam para as etapas seguintes, do desenvolvimento. Com a detecção atempada dos erros, consegue-se uma redução considerável de custos tanto ao nível de recursos como ao nível de encargos.

Sumariamente, podemos diferenciar o UML/OCL do Alloy, segundo as seguintes considerações:

- O UML/OCL tem um vasto conjunto de tipos primitivos, todos tipificados e bem definidos, o mesmo não acontece no Alloy.

Linguagem Corrente	Uma *Class* só pode do tipo: Class, Associations, Generalizations, Constraints, e Interfaces
OCL	Class seft.allContents-> forAll (c| c.oclIsKindOf(Class) or c.oclIsKindOf(Associations) or c.oclIsKindOf(Generalizations) or c.oclIsKindOf(Constraints) or c.oclIsKindOf(Interfaces))
ALLOY	all c: Class| c in allContens (Class + Associations + Generalizations + Constraints + Interfaces)

Tabela 4: Exemplo simples de OCL vs Alloy [32]

Linguagem Corrente	A operação *allParents* retorna um conjunto de todos elementos de *GeneralizableElements* herdado de *GeneralizableElement,* excluindo o próprio *GeneralizableElement.*
OCL	GeneralizableElement allParents: Set (GeneralizableElement); allParents=self.parent->union(self.parent.allparents)
ALLOY	all e: GeneralizableElement| e.allParents= e.+parent

Tabela 5: Exemplo de uma operação OCL vs Alloy [32]

- Ao nível da navegação, em Alloy existe a noção de escalares que podemos usar para construir os tipos dos conjuntos, o que não acontece no caso do OCL. Para aceder sintacticamente aos escalares utiliza o símbolo ".", enquanto nos conjuntos utiliza o "->". No entanto, as expressões o "." tem o mesmo significado tanto para escalares como para conjuntos, enquanto que o símbolo "->" significa o produto cartesiano.
- O UML distingue sintacticamente os atributos das relações, o Alloy não [Tabela 5].
- O Alloy possui um analisador automático, tem um alcance de pesquisa finito, que não retorna falsos negativos, mas a pesquisa é incompleta. No entanto,

segundo estudos realizados um âmbito pequeno pode ser muito útil na obtenção de erros existentes [40].

- Em Alloy, a descrição do sistema é baseada nos elementos que o compõem e na forma como estes interagem uns com os outros, não especificando o número possível de elementos que podem existir.
- Ao nível da herança, o UML/OCL e o Alloy utilizam diferentes paradigmas na forma como lidarem com a herança, com a performance e com a tipagem. Estas diferenças influenciam directamente o processo de tradução. No UML uma subclasse pode herdar ou especializar os atributos e as operações da sua superclasse ou de outras classes. Em Alloy, uma assinatura só pode estender de uma outra assinatura e os elementos da subassinatura são considerados como um subconjunto dos elementos da superassinatura. No entanto, a subassinatura não pode declarar um campo com o mesmo nome que a sua superassinatura declarou. Isto porque um campo da subassinatura não pode sobrepor-se ao campo de uma superassinatura. Uma das estratégias utilizadas é a utilização dos nomes correspondentes. É simples e intuitivo, mas introduz uma complexidade adicional. Isto porque temos de criar um campo com um nome único e alterar todas as suas referências anteriores. Também deve-se considerar as restrições impostas aos atributos da superclasse, uma vez que estas têm de ser propagadas para as suas subclasses.
- O UML permite múltiplas heranças e o Alloy não.
- No UML, a agregação e a composição são um tipo de associção que traduzem uma semântica precisa. Em Alloy não existe formas de distinguir, pois esta distinção é obtida através do auxílio dos quantificadores.
- Ao nível de filosofia das linguagens, o OCL é orientado à implementação enquanto Alloy é orientado à concepção.
- A sintaxe, semântica e gramática do OCL é mais verbosa do que as construções Alloy. A estrutura dos modelos Alloy é construída à base de átomos e relações. Toda a expressão denota uma relação. Uma função é apenas uma relação binária que mapeia os átomos de um lado da relação para o outro lado da mesma [Tabela 5].

- Em OCL, não se pode aplicar o conjunto de operadores a objectos da mesma classe. Neste caso, têm de utilizar quantificadores para auxiliar os cálculos. No Alloy, os tipos estão implícitos e associados ao domínio do modelo[32].

4.6 Conclusão

Neste capítulo, descrevemos a linguagem de modelação Alloy, assim como os seus conceitos básicos essenciais e modelação.

Apresentamos a sua ferramenta para análise automática de modelos, Alloy Analyzer. Realizamos uma comparação entre as linguagens UML e Alloy, realçando o facto, que o UML é sem dúvida uma das linguagens de modelação gráfica mais utilizada e com um potencial considerável mas ainda com alguma ambiguidade. A linguagem OCL colmata algumas destas ambiguidades mas é uma linguagem muito verbosa, por sua vez Alloy é conciso, declarável e automaticamente analisável. Nesta óptica, pensamos que o UML pode ser integrado com Alloy, isto porque as linguagens são compatíveis principalmente tendo em conta que o Alloy oferece de um analisador automático.

5 Mapeamento de Diagrama de Classes em Modelos Alloy

A linguagem UML é uma linguagem de modelação, não proprietária e sem metodologia própria, mas que permite aos analistas modelarem os seus sistemas em diagramas padronizados.

No seu âmago, o UML pretende padronizar as formas de modelação, tendo por objectivos a estruturação, a criação de um histórico de acções e facilitar a lógica da programação.

Existindo compatibilidade e bivalência entre a linguagem semi-formal de especificação UML e a linguagem formal de especificação Alloy, podemos juntar as suas mais-valias e obter modelos formais.

Neste capítulo focamo-nos na descrição e análise do mapeamento dos diagramas de classe UML para modelos Alloy. Na secção 5.1 apresentamos a motivação para o processo de refinamento; na secção 5.2 apresentamos o mapeamento de elementos construtores do UML para elementos construtores de Alloy; na secção 5.3 especificamos o conceito de rastreabilidade e sua importância no nosso contexto; na secção 5.4, apresentamos um exemplo genérico do mapeamento de um diagrama de classes UML para um modelo Alloy; na secção 5.5 apresentamos a organização da estrutura dos módulos do sistema de verificação; e na secção 5.6 finalizamos este capítulo.

5.1 Motivação

Como referido, o conceito de refinamento que servirá de motor para a definição de um conjunto de regras formais cujo intuito é garantir a preservação semântica entre modelos.

O conjunto de regras de refinamento é a base para a técnica de verificação poder averiguar se os modelos são equivalentes, ou não. Desta dedução, podemos inferir se estes são, ou não, consistentes.

Neste âmbito, o conceito de refinamento é relativo ao relacionamento entre dois modelos (Figura 12: Refinamento entre dois diagramas de classesFigura 12). Assim, neste contexto definimos refinamento como:

> "um componente aperfeiçoado que é considerado '*melhor*', podendo ser utilizado em lugar do seu original, mas sem poder modificar as propriedades do modelo original, do sistema."

Figura 12: Refinamento entre dois diagramas de classes

Ao nível de âmbito de actuação e de forma generalista, podemos dizer que o refinamento permite:

- Manter todos conceitos básicos iniciais, não sendo possível a sua eliminação, mas somente a sua renomeação ou transformação;
- Adicionar informação a um modelo aumentando o seu grau de detalhe; e
- Analisar e verificar situações inexploradas que ajudam à prevenção de erros subtis.

Para automatizar a análise do refinamento, este tem de ser desenvolvido através de modelos formais. Neste sentido, para implementar a verificação automática do refinamento tivemos a necessidade de utilizar uma linguagem de especificação formal Alloy.

Na Figura 1 apresentamos, de forma esquematizada, a metodologia do nosso trabalho. Devemos notar que em ambas as representações dos diagramas não há modificação das propriedades estruturais do sistema.

Como referido, já existem outros mapeamentos de UML para Alloy [17]. No entanto, neste trabalho iremos usar uma abordagem diferente.

5.2 Representação do Metamodelo UML em Alloy

Na abordagem orientada a objectos, a modelação de uma estrutura de um sistema de informação consiste na identificação das suas classes e das suas relações. Isto porque,

na sua totalidade, o diagrama de classes representa o domínio do problema ou a visão geral do sistema.

O *MetamodelUML2Alloy* é metamodelo criado para facultar o mapeamento. Assim, através das assinaturas conseguimos criar estruturas com capacidades matemáticas, essenciais à correcção e rigor da representação e modelação da informação, em qualquer tipo de sistemas.

Foi nossa preocupação que as assinatura s conseguissem capturar e expressar "*a forma e como*" revelar toda a informação, assim como a sua semântica.

Na Figura 13 apresentamos um subconjunto do metamodelo UML representado em Alloy.

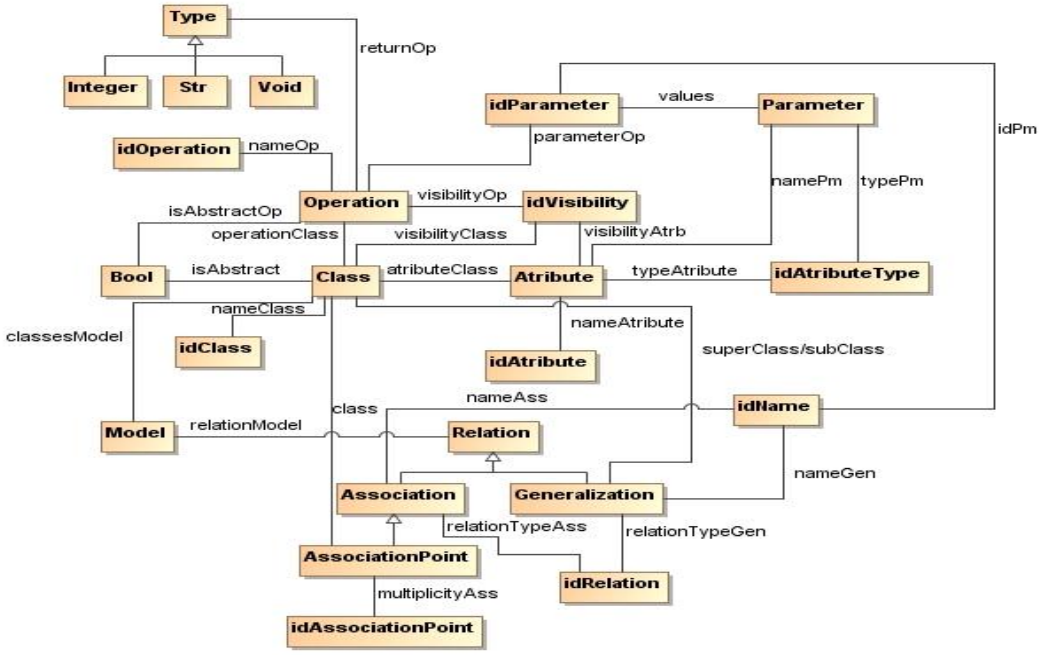


Figura 13: Subconjunto do metamodelo UML representado em Alloy

Elemento Type

A assinatura base do M*etamodeloUML2Alloy* é assinatura *Type*. Esta tem por objectivo fornecer a definição para os tipos de dados, que podem ser ou não tipificados, e que posteriormente são utilizados na definição dos modelos. Esta assinatura serve de pilar a toda a hierarquia modelar, da nossa arquitectura.

Elemento Model

Um modelo é usado para representar toda a estrutura de um sistema, dando um panorama geral do mesmo. Neste âmbito e com o intuito de traduzir toda a informação de um diagrama, temos a assinatura *Model*. Esta é composta pelos seguintes campos:

Descrição	Alloy

• classesModel – identifica todas as classes que compõem o diagrama.
• relationsModel – identifica todas as relações que compõem o diagrama.
	abstract sig **Model** {
	classesModel : set Class,
	relationsModel : set Relation
	}

Tabela 6: Elemento Model

Elemento Class

Em UML, a classe tem por objectivo providenciar uma entidade composta por um conjunto de propriedades estruturais e responsabilidades capazes de a caracterizar. Todas as instâncias da classe partilham os mesmos atributos, operações, relações e semântica, distinguindo-se somente no conteúdo da informação.

Assim, ao nível de propriedades estruturais, uma classe deve possuir: um nome, qual o seu estado (se é ou não abstracta), qual a sua visibilidade, quais os seus atributos e quais as suas operações.

Neste âmbito, para a tradução de uma classe do diagrama de classes UML para um modelo Alloy, temos a assinatura *Class*, composta pelos seguintes campos:

Descrição	Alloy

• nameClass – identifica o nome unívoco da classe. Este deve ser único e descritivo, perante todo o sistema.
• *isAbstract* – identifica qual o seu estado. Por defeito, este campo é declarado com o valor booleano "*False*". Quando a classe é abstracta, a sua declaração é feita com o valor "*True*".
• visibilityClass – identifica qual a visibilidade da classe, permitindo desta forma determinar qual é o seu acesso.
• atributesClass – identifica o conjunto dos atributos pertencentes à classe.
• operationsClass - identifica o conjunto das operações pertencentes à classe.
	abstract sig **Class** {
	isAbstract: one Bool,
	visibilityClass: one idVisibility,
	nameClass: one idClass,
	atributesClass : set Atribute,
	operationsClass : set Operation }
	{
	isAbstract=True or
	isAbstract=False
	visibilityClass=publico or
	visibilityClass=protegido or
	visibilityClass=privado
	}

Tabela 7: Elemento Class

Elemento Atribute

Os atributos são propriedades estruturais internas de uma classe. Estes permitem descrever detalhadamente as características da classe, assim como o tipo de valores possíveis que podem ter, ao longo da sua utilização.

O estado de uma classe pode ser definido pelos valores atribuídos aos seus atributos. Nesta óptica, a assinatura *Atribute* é a responsável por descrever um atributo. Esta é composta pelos seguintes campos:

Descrição	Alloy

• nameAtribute - identifica o nome unívoco de um atributo, de uma classe.
• typeAtribute – identifica o tipo do domínio do atributo.
• visibilityAtrib - identifica qual a visibilidade do atributo.
	`abstract sig Atribute {`
	`  visibilityAtribute: one idVisibility,`
	`  nameAtribute: one idAtribute,`
	`  typeAtribute: one idTypeAtribute`
	`}`

Tabela 8: Elemento Atribute

Elemento Operation

As operações são as acções de consulta ou alteração realizadas a entidade, num sistema. O comportamento de uma classe é definido pelo conjunto destas acções que podem ser originar uma mudança de estado. Desta forma, o comportamento é determinado pelos serviços que a classe disponibiliza às outras classes.

Uma operação deve possuir: um nome, uma visibilidade, um tipo, uma sequência de parâmetros e um valor de retorno. Um parâmetro é composto por zero ou mais atributos tendo estes a indicação do seu tipo de dados.

A assinatura *Operation* é a responsável por descrever as operações. Esta é composta pelos seguintes campos:

Descrição	Alloy

• nameOp - identifica o nome da operação, de uma classe.
• *isAbstract* – identifica o estado de uma operação. Por defeito, este campo é declarado com o valor booleano "*False*". Quando temos uma operação abstracta, a sua declaração é feita com o valor "*True*".
	`abstract sig Operation {`
	`  isAbstractOp: one Bool,`
	`  visibilityOp: one idVisibility,`
	`  nameOp: one idOperation,`
	`  parameterOp : set idParameter,`
	`  returnOp : one idType`
	`}`

• visibilityOp – identifica qual a visibilidade da operação. • parametersOp – identifica os parâmetros de entrada da operação. • returnOp – especifica o tipo de retorno da operação.	

Tabela 9: Elemento Operation

Elemento idParameter

Os parâmetros são especificações de atributos e respectivos tipos que fazem parte de uma operação. Uma operação com o mesmo nome pode ter diferente número de parâmetros. A assinatura *idParameter* é a responsável por descrevê-los. Esta é composta pelos seguintes campos:

Descrição	Alloy

• idPm - identifica o nome do parâmetro.
• values – identifica o conjunto dos atributos e respectivos tipos pertencentes ao parâmetro.
	`abstract sig idParameter{`
	`idPm:one idName,`
	`values: set Parameter`
	`}`

Tabela 10: Elemento idParameter

Elemento Parameter

Este elemento estende o anterior identificando o conjunto de atributos e seus respectivos tipos. A assinatura *Parameter* é composta pelos seguintes campos:

Descrição	Alloy

• namePm – identifica os atributos pertencentes ao parâmetro.
• typePm – identifica o tipo dos atributos pertencente ao parâmetro.
	`abstract sig Parameter {`
	`namePm: set Atribute,`
	`typePm: one idTypeAtribute`
	`}`

Tabela 11: Elemento Parameter

Elemento Relation

Com referido, em UML, os relacionamentos entre classes expressam a forma como estas interagem entre si. Uma relação pode ser de dois tipos distintos: associação ou generalização. Numa associação temos uma relação entre as classes, na generalização temos uma ligação de herança.

Elemento Association

Uma associação pode assumir os seguintes tipos distintos: associação, agregação ou composição. Na definição de uma associação temos o tipo, a visibilidade, o nome do vértice inicial, o nome do vértice final e a multiplicidade. A assinatura *Association* descreve as relações deste tipo. Esta é composta pelos seguintes campos:

Descrição	Alloy

• relationTypeAss – identifica o tipo da associação. Este campo está restringido aos seguintes valores: "*associa, agrega ou compoe*".
• visibilityAss - identifica a visibilidade da associação.
• start – identifica o vértice inicial da associação.
• end - identifica o vértice final da associação.
	`abstract sig Association extends Relation {`
	`visibilityAss: one idVisibility,`
	`relationTypeAss : one idRelation,`
	`start : one AssociationPoint,`
	`end : one AssociationPoint }`
	`{`
	`visibilityAss=publico or`
	`visibilityAss=protegido or`
	`visibilityAss=privado`
	`relationTypeAss=associa or`
	`relationTypeAss=agrega`
	`}`

Tabela 12: Elemento Association

Elemento AssociationPoint

A *Association Point* estabelece uma associação entre duas classes. Ela é composta pelos seguintes campos:

Descrição	Alloy

• class – identifica a classe em questão.
• multiplicityAss – identifica o numero de objectos que podem estar associados a cada vértice da associação. Este campo está restrito aos seguintes valores: zero, um ou muitos.
	`abstract sig AssociationPoint extends Relation {`
	`class: one Class,`
	`multiplicityAss: one idAssociationPoint`
	`}`

Tabela 13: Elemento AssociationPoint

Elemento Generalization

A generalização é um outro tipo distinto de uma relação, que se distingue por ter um elemento geral e um ou mais elementos mais específicos, permitindo representar o

conceito de herança. Assim, temos a assinatura *Generalization* composta pelos seguintes campos:

Descrição	Alloy

• relationTypeGen - identifica o tipo generalização. Este campo está restrito ao valor "*generaliza*".
• nameGen - identifica o nome da generalização.
• superClass – identifica a classe geral (superclasse).
• subclass – identifica as subclasses da classe geral.
	`abstract sig Generalization extends Relation {`
	`relationTypeGen: one idRelation,`
	`subClass: set Class,`
	`superClass: one Class }`
	`{`
	`relationTypeGen=generaliza`
	`}`

Tabela 14: Elemento Generalization

5.3 Rastreabilidade

Do modelo original não é possível inferir directamente toda informação para o modelo destino. Neste âmbito, recorremos ao conceito de rastreabilidade de forma a mapear a informação modificada relevante, do modelo original para o modelo destino.

A rastreabilidade serve de guia ao processo de verificação do refinamento, permitindo a correspondência de um elemento de um modelo noutro elemento correspondente noutro modelo.

No âmbito deste trabalho, utilizamos este conceito com base na forma de *como* fazemos corresponder um elemento a outro elemento nos diferentes modelos. Nesta perspectiva, o objectivo é permitir:

- Compreender a origem dos elementos;
- Verificar se todos os elementos do sistema estão correctamente implementados.

De forma abrangente, a rastreabilidade permite-mos compreender e organizar a forma como as informações fornecidas sobre os elementos são convertidas.

5.3.1 Metamodelo da Rastreabilidade

Como já mencionado, o sistema de rastreabilidade proposto tem por objectivo proporcionar a correspondência de elementos. Na Figura 14 apresentamos o metamodelo da rastreabilidade proposto:

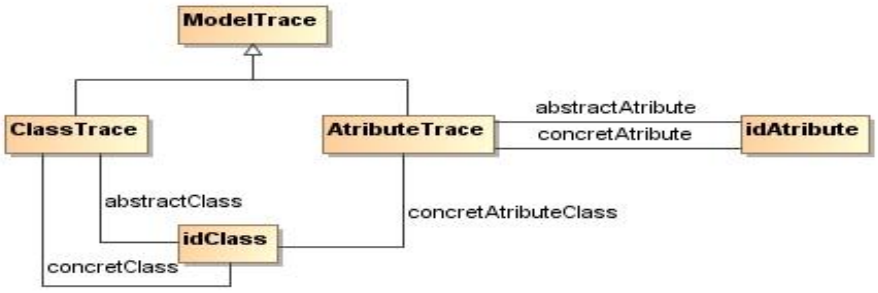


Figura 14: MetaTraceability

A rastreabilidade tem duas conjunturas distintas: o das classes e o dos atributos. A rastreabilidade das classes permite fazer a correcta correspondência das classes do modelo original para as classes do modelo destino. Este mapeamento é necessário porque o Alloy impõe que todas as assinaturas existentes no domínio em análise tenham um nome único. Assim, em todo o modelo não podem existir assinaturas com o mesmo nome.

No caso dos atributos, a correspondência tem dois cenários distintos:

- quando um atributo do modelo original é decomposto em vários atributos no modelo destino;
- quando um atributo do modelo original é transformado numa nova classe que pode, ou não, ter vários atributos no modelo destino. Este cenário pressupõem a criação de um relacionamento do tipo associação, onde a classe do modelo original identifica o *todo* e a classe gerada da transformação significa *parte* desse *todo*.

Para a codificação da rastreabilidade, temos as assinatura s *ModelTrace*, *ClassTrace* e *AtributeTrace*, cujo objectivo é identificar a correspondência entre os elementos do modelo original no modelo destino.

Elemento ModelTrace

A assinatura *ModelTrace* é a estrutura que representa a informação relativa às classes e aos atributos que individualmente servem de elementos correspondentes (refinados) aos diferentes diagramas. Esta é composta pelos seguintes campos:

Descrição	Alloy

• *allClassesTrace* – identifica os átomos de correspondência das classes.
• *allAtributesTrace* – identifica os átomos de correspondência dos atributos.
	`abstract sig ModelTrace{`
	`  allClassesTrace: set ClassesTrace,`
	`  allAtributesTrace:set AtributesTrace`
	`}`

Tabela 15: Elemento ModelTrace

Elemento ClassTrace

A assinatura *ClassTrace* é a estrutura que representa a informação relativa à correspondência de uma classe do modelo original no modelo destino. Esta é composta pelos campos:

Descrição	Alloy

• *abstractClass*: identifica a classe do modelo original que sofrerá uma transformação.
• *concretClass*: identifica a classe ou conjunto de classes de correspondência originadas pelo refinamento anterior, no modelo destino.
	`abstract sig ClassesTrace extends ModelTrace{`
	`abstractClass: one idClass,`
	`concretClass: set idClass`
	`}`

Tabela 16: Elemento ClassTrace

Elemento AtributeTrace

A assinatura *AtributeTrace* é a estrutura que representa a informação relativa à correspondência de um atributo do modelo original no modelo destino. Esta é composta pelos campos:

Descrição	Alloy

• *abstractAtributeRefine*: identifica o atributo do modelo original.
• *concretAtributeRefine*: identifica um ou mais atributos a serem transformados.
• *concretClassRefine*: identifica o nome de correspondência da classe transformada no modelo destino.
	`abstract sig AtributesTrace extends ModelTrace{`
	`abstractAtributeTrace: one idAtribute,`
	`concretAtributeTrace: set idAtribute,`
	`concretAtributeClassTrace: set idClass`
	`}`

Tabela 17: Elemento AtributeTrace

Na tabela de rastreabilidade temos os seguintes cenários possíveis:

- Renomeação do nome da classe do modelo original para um novo nome diferente no modelo destino.

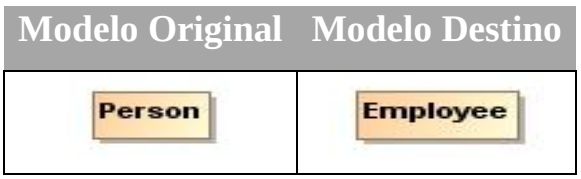


Figura 15: Rastreabilidade – Renomeação de uma classe

- Renomeação do nome da classe do modelo original transformado em várias classes no modelo destino.

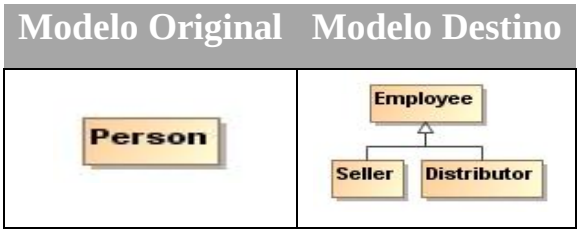


Figura 16: Rastreabilidade – Transformação de uma classe em várias classes

- Alteração de um atributo do modelo original num ou mais atributos no destino.

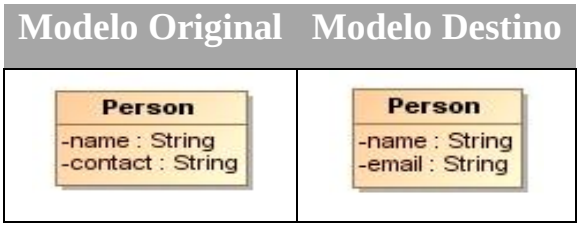


Figura 17: Rastreabilidade – Renomeação de atributos

- Alteração de um atributo do modelo original numa nova classe com novos atributos no modelo destino.

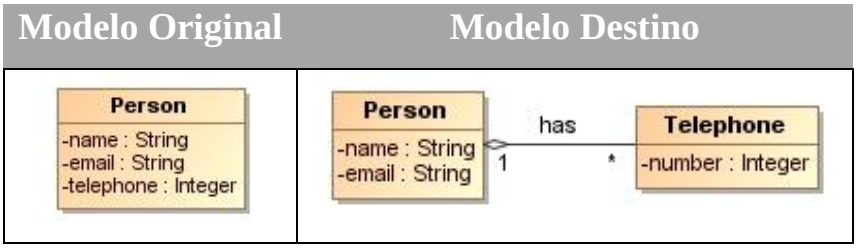


Figura 18: Rastreabilidade – Transformação de um atributo numa nova classe

Em Alloy, a tabela rastreabilidade é implementada pelo módulo "*tableTraceability*", que estende o módulo "*MetaTraceability*".

Este módulo tem por objectivo instanciar todos os elementos de correspondência. Este é essencial na fase de análise. Pois, é a tabela de rastreabilidade que permite fazer a correspondência dos elementos refinados entre os modelos.

5.4 Tradução de Modelos para Alloy

Neste item, procederemos à descrição do mecanismo de mapeamento dos modelos, mapeando todos elementos dos diagramas de classes UML em assinaturas Alloy. Este mapeamento é feito segundo a estrutura definida no "*MetamodelUml2Alloy*".

A linguagem Alloy é influenciada pelas notações da modelação orientada por objectos, tornando-a assim mais fácil e compreensível a classificação de objectos e suas propriedades.

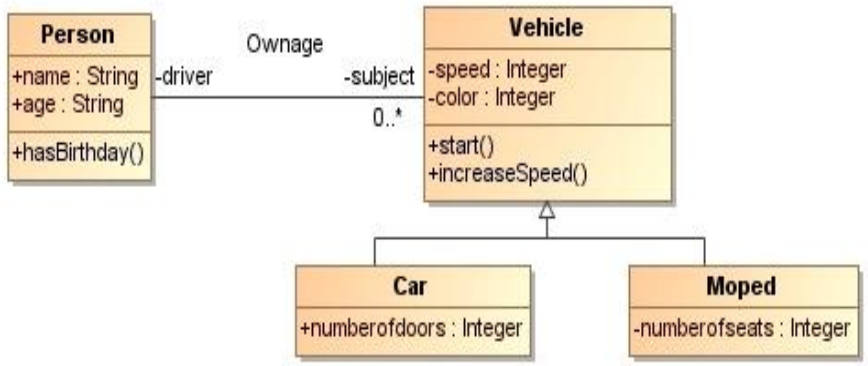
Seguindo o paradigma das linguagens orientadas por objectos, Alloy suporta a organização de um modelo em módulos, onde cada módulo pode aceder directamente ao conteúdo de um outro módulo. Nesta perspectiva, cada diagrama é traduzido num módulo único, acedendo às estruturas de outros módulos.

Com o intuito de tornar pragmático o mecanismo de tradução e facilitar a sua compreensão, passamos apresentar o mapeamento de um diagrama de classes UML para um modelo Alloy.

Uma das limitações do Alloy é que o nome de uma assinatura tem de ser único. Para superar esta limitação, optamos por no modelo original, no final de cada elemento adicionamos a etiqueta "*A*", e no modelo destino, no final de cada elemento adicionamos a etiqueta "*C*".

O seguinte exemplo ilustra um diagrama de classes UML abstracto e sua respectiva tradução em Alloy.

Diagrama Classe UML



Modelo Alloy

```
sig PersonVehicleAbs extends Model {}
{       classesModel =  PersonA + VehicleA + CarA + MopedA
        relationsModel=  PersonVehicleA + VehicleGenA
}
```

(classesModel)

```
sig PersonA extends Class{}
{       isAbstract=False
        visibilityClass=publico
        nameClass=Person
        atributesClass = ageA + nameA
        operationsClass=  hasBirthdayA
}
```

(relationsModel)

```
sig PersonVehicleAssA extends Association{}
{       visibilityAss= public
        nameAss= drive
        associationTypeAss= associa
        start= driverA
        end= subjectA }
sig driverA extends AssociationPoint{}
{       class= PersonA
        multiplicityAss= muitos      }
sig subjectA extends AssociationPoint{}
{       class= VehicleA
        multiplicityAssMax=  zero    }
```

	*sig ageA extends **Atribute**{}* *{ visibilityAtrib= publico* *nameAtrib=age* *typeAtribute = tipoINT* *}*	*Sig VehicleGenA extends **Generalization*** *{}* *{ relationTypeGen= generaliza* *subclass= MopedA + CarA* *superClass= VehicleA }*
	*sig hasBirthdayA extends **Operation**{}* *{ isAbstractOp=False* *visibilityOp=publico* *returnOp = Str* *nameOp= hasBirthdayA* *parametersOp.idPm = none* *parametersOp.values.namePm= none* *parametersOp.values.typePm= none* *}*	

Tabela 18: Tradução de um Diagrama de Classes UML em código Alloy [30]

Em Alloy, para que um modelo seja considerado válido, não pode violar nenhuma restrição que seja definida através dos seus factos. Um modelo é considerado inconsistente se não existir uma instância que satisfaça todas as restrições estipuladas.

Sendo assim, para ter-se uma instância válida, por exemplo, de uma classe é necessário que sejam satisfeitas todas as restrições a si impostas.

O domínio de análise de um modelo é composto pelas assinaturas referenciadas na *classesModel*, na *relationsModel*, na *atributesClass* e na *operationsClass*.

5.5 Módulos do Sistema de Verificação

Neste item apresentamos um conjunto de módulos essenciais ao processo de análise de refinamento. Os principais módulos usados são:

- Type – estrutura do tipo básico de dados.
- MetamodelUML2Alloy – estrutura de dados para a representação de diagramas de classes. Tem por objectivo fornecer uma sintaxe para a modelação dos principais elementos que compõem um diagrama de classes UML, permitido assim que seja possível a tradução destes em modelos formais Alloy.
- UniqueName – módulo que permite a declaração dos nomes de correspondência, segundo os seus tipos.
- MetamodelTraceability – estrutura de dados que permite descrever a correspondência de um elemento do modelo original, num ou mais elementos no modelo destino.

- TableTraceability – módulo que permite a instanciação da tabela de rastreabilidade.
- FunTrace – módulo que fornece um conjunto de funções auxiliares que nos permitem manipular a tabela de rastreabilidade.
- Model – módulos que permitem a instanciação dos modelos que serviram de dados de entrada.
- MapModel – módulo que permite a criação de um elo de ligação entre os dois modelos.
- RuleRefinement – sistema de regras de refinamento. Tem por objectivo criar o sistema de regras de refinamento proposto. Este é decomposto em várias categorias, segundo o seu tipo de construtor: classe, atributo, operação ou relacionamento.

Na Figura 19 apresentamos a decomposição hierárquica dos módulos em Alloy:

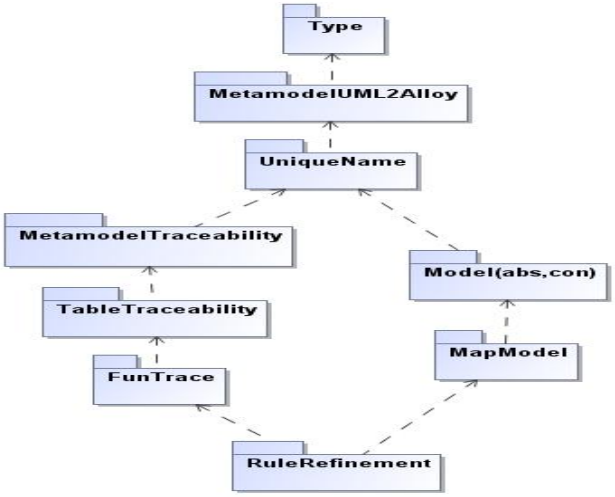


Figura 19: Decomposição hierárquica dos módulos Alloy

5.6 Conclusão

Neste capítulo foi apresentado a formalização de um diagrama de classes UML num modelo Alloy através do *MetamodelUML2Alloy*.

Os trabalhos OhCircus e UML2ALLOY forneceram algumas das directrizes para o sistema proposto. No entanto, distingue-se na forma como se faz a especificação das regras.

A utilização de um metamodelo permitiu-nos a definição das regras de transformação entre modelos de linguagens diferentes (uma linguagem semi-formal e outra formal). Este mapeamento possibilitou a tradução directa dos elementos entre os diferentes modelos.

6 Regras de Refinamento

Neste capítulo, descrevemos detalhadamente o sistema de regras de refinamento proposto.

Como referido, este sistema tem como intuito a produção de um conjunto de regras que possibilita a verificação semântica dos modelos após o processo de refinamento.

Na secção 6.1 apresentamos o princípio de transformação assim como o enquadramento abstracto do mecanismo de refinamento; na secção 6.2 enunciamos um conjunto de transformações que nos permitem deduzir a existência de refinamento; na secção 6.3 descrevemos as regras do sistema de regras de refinamento proposto; na secção 6.4 descremos o processo de validação e respectivo algoritmo de implementação que nos permite deduzir a existência e o tipo de refinamento; e na secção 6.5 apresentamos as conclusões deste capítulo.

6.1 Motivação

O processo de transformação centraliza-se na forma de *como* devem ser mapeados os elementos do modelo original para os elementos de um modelo final. Isto é, o modelo final tem de respeitar rigorosamente o(s) principio(s) de transformação(ões) dos construtores do modelo original.

Os niveis abstracção podem ser categorizados em dois niveis distintos: o nivel horizontal e o nivel vertical. No nivel horizontal, as transformações dão-se ao mesmo nivel hierárquico isto é, uma classe pode ser renomeada ou um atributo pode dar origem a novos atributos, enquanto que no nivel vertical, as transformações dão-se a diferentes níveis hierárquicos, isto é, a transformação de um atributo numa nova classe deu origem a uma nova relação logo uma novo nível hierárquico.

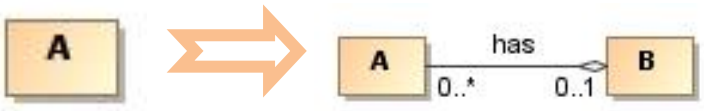
O refinamento é o mecanismo que se enquadra no nivel vertical. Isto porque no processo de desenvolvimento de software uma especificação pode ser gradualmente aperfeiçoada, originando assim diferentes niveis de abstraçao (adição de detalhe a cada nova especificação).

6.2 Transformações de Refinamento

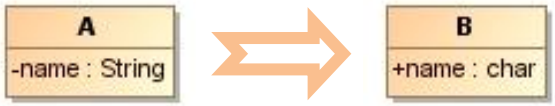
Sendo o refinamento um mecanismo que permite adição de detalhe no modelo destino, isso implica que são necessários realizar um número considerável de acções, possibilitando-nos obter um modelo, o mais próximo possível do modelo de implementação.

De forma generalista [9], apresentamos as seguintes as regras que nos permitem deduzir a existência de refinamento.

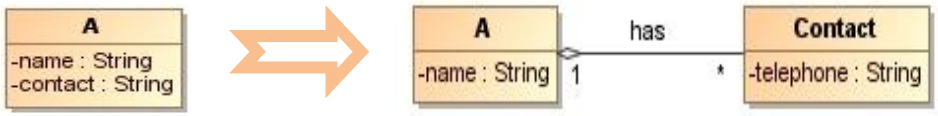
- Adição de novas classes, novos atributos, novos métodos e novos relacionamentos:



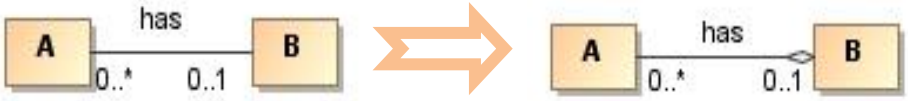
- Alteração das propriedades intrínsecas de classes, atributos ou métodos:



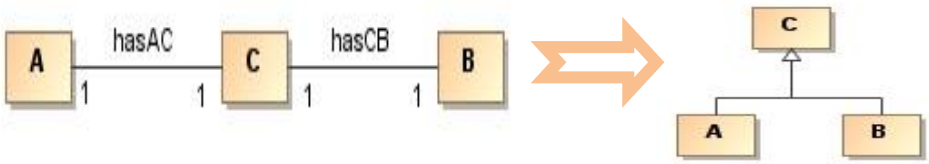
- Transformação de atributos em novos atributos ou classes:



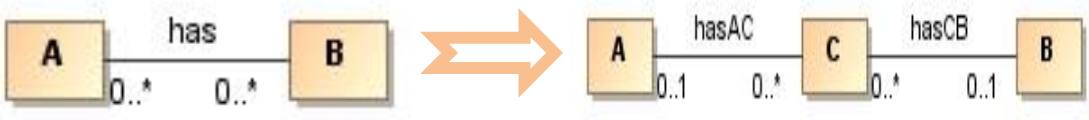
- Transformação de uma associação numa agregação:



- Transformação de uma associação numa generalização:



- Transformação de associações de muitos-para-muitos:



Na Figura 20 apresentamos a classificação das regras de refinamento propostas.

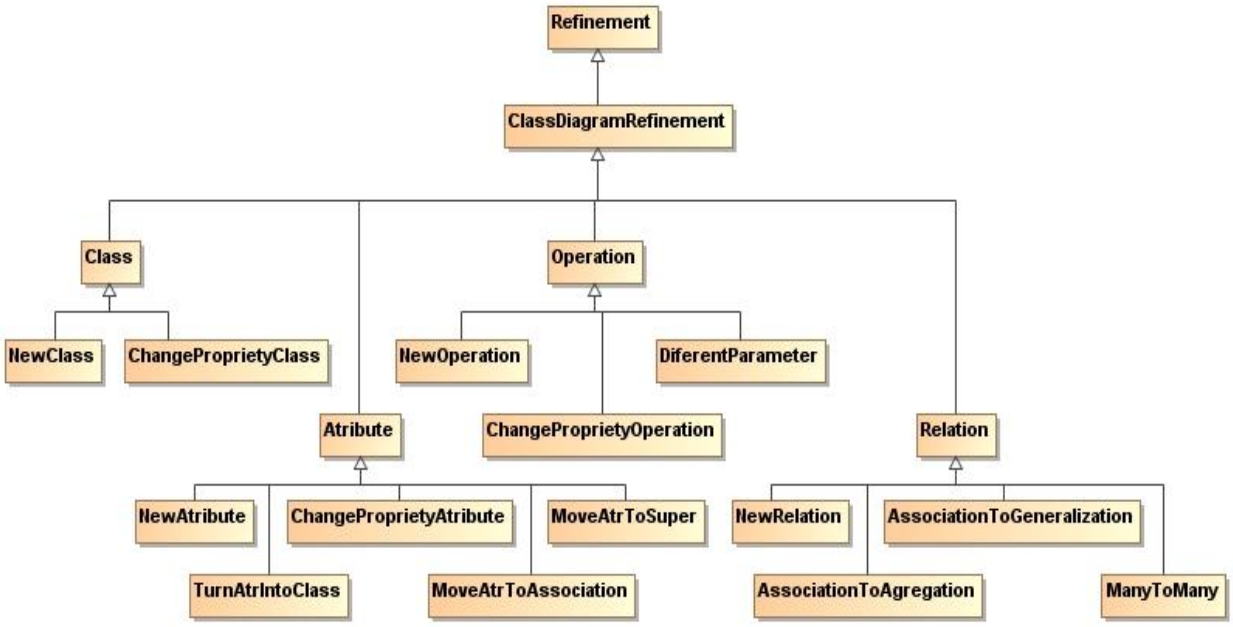


Figura 20: Classificação das regras de refinamento

A taxonomia apresentada pretende definir um conjunto de alterações aceites para um conjunto de restrições que devem ser respeitadas de forma a garantir a existência de refinamento.

As restrições estão descritas em forma de asserções em Alloy, permitindo validar e garantir a correcção das transformações realizadas, no modelo destino.

6.3 Descrição das Regras de Refinamento

Neste item descrevemos detalhadamente as regras de refinamento. As regras de refinamento vão permitir verificar algumas propriedades algébricas do modelo original e do modelo destino através da análise automática usando o Alloy Analyzer.

Com o intuito de facilitar a compreensão das regras, estas estão implementadas em módulos distintos.

Ao nível de estruturação, criamos quatro categorias distintas de regras: regras de Classe, regras de Atributo, regras de Operação e regras de Relacionamento.

Pautando-nos pelas boas práticas de modelação, estruturamos a descrição das regras da seguinte forma:

- nome unívoco da regra que a identifica perante o sistema;
- objectivo que sumariza a sua finalidade;
- descrição detalhada;
- propriedade estrutural;

- exemplo ilustrativo da sua aplicação; e
- o nome do predicado ou da asserção em código Alloy, apresentado no anexo A.

A tabela de rastreabilidade é uma fonte de informação essencial ao processo de refinamento. A sua relevância é demonstrada nos casos de renomeação do nome de uma classe, da transformação de um atributo numa classe, entre outros. Assim, sempre que necessário o modelo destino deverá incluir informação sobre a tabela de rastreabilidade.
A partir de agora vamos designar o modelo original como modelo original e o modelo destino como modelo destino.

6.3.1 Regras sobre Classes

Neste item, apresentamos as regras relativas à categoria da classe. Na Tabela 19 temos o nome e uma descrição sintetizada das regras desta categoria.

Regra	Função	Descrição Síntese
Regra1a	Adição de Classes.	Determinar as novas classes do modelo destino.
Regra1b	Alteração das características das classes.	Verificação de alteração das características de uma classe.

Tabela 19: Regras de refinamento de classes

Regra1a – Adição de Classes

Objectivo:

Identificar quais as novas classes existentes no modelo destino, que referenciadas na tabela de rastreabilidade não estão no modelo original.

Descrição:

Todas classes existentes no modelo original juntamente com as referências da tabela da rastreabilidade têm de estar contidas no modelo destino, para que este modelo seja considerado válido.

Assim, esta regra devolve as novas classes existentes no modelo destino. Quando temos um retorno nulo isso significa que não existem novas classes, após o processo de transformação.

Propriedade:

$$\forall abs, con \in Model.classesModel, traceAbs, traceCon \in Trace.allClass :$$
$$(abs.nameClass - traceAbs.classAbs) \cup traceCon.classCon \subseteq con.nameClass$$

Exemplo:

Modelo Original	*Modelo Destino*
	
Alloy	

sig nameA extends Atribute {}{
visibilityAtribute= privado
nameAtribute=name
typeAtribute=tipoSTR }
sig getNameA extends Operation{}{
isAbstract=False
visibilityOp=publico
returnOp = Str
nameOp=getName
parameterOp=param }
sig PersonA extends Class {}{
isAbstract=False
visibilityClass=publico
nameClass=Person
atributesClass = nameA
operationsClass = getNameA}
	sig nameC extends Atribute {}{
	visibilityAtribute=privado
	nameAtribute=name
	typeAtribute=tipoSTR }
	sig getNameC extends Operation{}{
	isAbstract=False
	visibilityOp=publico
	returnOp = Str
	nameOp=getName
	parameterOp=param }
	sig EmployeeC extends Class {}{
	isAbstract=False
	visibilityClass=publico
	nameClass=Employee
	atributesClass = nameC
	operationsClass = getNameC }

Rastreabilidade: Person->Employee	

sig tableTrace extends ModelTrace{}{
allClassesTrace= PerEmp
allAtributesTrace= none }
sig PerEmp extends ClassesTrace {}{
abstractClass= Person
concretClass= Employee }	

Tabela 20: Exemplo da regra de adição de classes

Código Alloy:

assert NewClass{} *– ver anexo A.5*

Regra1b – Alteração de propriedades de uma classe

Objectivo:

Verificar a existência de alterações das propriedades estruturais iniciais nas classes do modelo original em relação ao modelo destino.

Descrição:

Após um processo de refinamento, a adição de detalhes pode-se manifestar em alterações das propriedades estruturais das classes do modelo original. Assim numa classe, deve-se verificar as seguintes propriedades:

- Abstracção;
- Identificador; e
- Visibilidade.

Ao nível da propriedade que identifica se uma classe é abstracta, por defeito, esta é instanciada com o valor False. No entanto, quando esta é instanciada com o valor True, significa que a classe não pode ser directamente instanciada e fornece propriedades e funcionalidades às restantes classes que a estendam. Assim, no modelo destino deve-se apresentar como uma superclasse.

Ao nível da propriedade que identifica o nome, esta representa o elo de ligação em Alloy entre as assinaturas dos diferentes modelos.

Como referido, o Alloy não permite nomes de assinaturas repetidos no mesmo âmbito de análise, logo optamos por designar todas as assinaturas do modelo original com a etiqueta "*A*" e as do modelo destino com a etiqueta "*C*".

Ao nível da propriedade que identifica a visibilidade da classe, a sua alteração não é trivial e pode gerar algum tipo de controvérsia. Isto porque a sua utilização depende do contexto da sua utilização.

Propriedade:

$$\forall abs, con \in Model.classesModel, traceAbs, traceCon \in Trace.allClass :$$
$$(abs.nameClass \neq con.nameClass) \vee$$
$$(abs.nameClass = trace.Abs.classAbs \wedge con.nameClass = traceCon.classCon) \vee$$
$$(abs.isAbstract \neq con.isAbstract) \vee (abs.visibility \neq con.visibility)$$

Exemplo:

Modelo Original	*Modelo Destino*
	
Alloy	

sig nameA extends Atribute {}{
visibilityAtribute=publico
nameAtribute=name
typeAtribute=tipoSTR }
sig getNameA extends Operation{}{
isAbstract=False
visibilityOp=publico
returnOp = Str
nameOp=getName
parameterOp=param }
	sig nameC extends Atribute {}{
	visibilityAtribute=privado
	nameAtribute=name
	typeAtribute=tipoSTR }
	sig getNameC extends Operation{}{
	isAbstract=False
	visibilityOp=publico
	returnOp = Str
	nameOp=getName
	parameterOp=param }

sig PersonA extends Class {}{ isAbstract=False visibilityClass=publico nameClass=Person atributesClass = nameA operationsClass = getNameA}	sig EmployeeC extends Class {}{ isAbstract=False visibilityClass=publico nameClass=Employee atributesClass = nameC operationsClass = getNameC }
Rastreabilidade: Person->Employee	
…. sig PerEmp extends ClassesTrace {}{ abstractClass= Person concretClass= Employee }	

Tabela 21: Exemplo da regra de alteração de propriedades de uma classe

Código Alloy:

assert ChangeProprietyClass{} *– ver anexo A.5*

6.3.2 Regras sobre Atributos

Neste item, apresentamos as regras sobre os atributos. Na Tabela 22 temos o nome e uma descrição sintetizada das regras desta categoria.

Regra	Função	Descrição Síntese
Regra2a	Adição de Atributos.	Determinar os novos atributos no modelo destino.
Regra2b	Alteração das propriedades dos atributos.	Verificar se o mesmo atributo, em modelos diferentes tem as mesmas propriedades.
Regra2c	Mover atributo de subclasses para a superclasse.	Verificar se existe um atributo com a mesma identificação nas diferentes subclasses da mesma superclasse.
Regra2d	Mover atributo de uma classe para outra a si associada.	Verificar se um atributo que pertence a uma classe ao modelo original está numa outra classe, associada à anterior, no modelo destino.
Regra2e	Transformar um atributo numa classe.	Verificar se um atributo do modelo original é transformado numa classe no modelo destino.

Tabela 22: Regras de refinamento do atributo

Regra2a – Adição de Atributos

Objectivo:

Identificar quais os novos atributos existentes no modelo destino que juntamente com as referências da tabela da rastreabilidade não estão contidos no modelo original.

Descrição:

Com já referido, um atributo descreve uma propriedade estrutural de uma classe. Neste âmbito, começamos por verificar se todos os atributos existentes no modelo original juntamente com os atributos referenciados na tabela de rastreabilidade estão contidos no modelo destino.

Após o processo de transformação, um atributo do modelo original pode originar, no modelo destino, um ou mais novos atributos, ou ainda pode gerar novos atributos e novas classes. É o caso exemplificado nas Tabelas 21 e 22, onde o atributo "*contact*" da classe "*Person*" do modelo original é refinado gerando dois cenários distintos.

No primeiro cenário, no modelo original a classe "*Person*" tem dois atributos. O atributo "*contact*" é mapeado no atributo "*email*" no modelo destino.

Exemplo - Cenário 1:

Modelo Original	Modelo Destino
	

Rastreabilidade: contact -> Contact
`sig tableTrace extends ModelTrace{}{`
`allClassesTrace= PerEmp`
`allAtributesTrace= contactAtributesClass }`
…..
`sig contactAtributesClass extends AtributesTrace{}{`
`abstractAtributeTrace=contact`
`concretAtributeTrace = email`
`concretAtributeClassTrace = none }`	

Tabela 23: Adição de novos atributos

No segundo cenário, atributo "*contact*" é refinado originando uma nova classe "*Telephone*" com um novo atributo "*number*" no modelo destino.

Exemplo – Cenário 2:

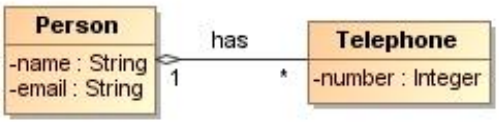
Modelo Original	*Modelo Destino*
	
Rastreabilidade: contact -> Contact …. sig contactAtributesClass extends AtributesTrace{}{ abstractAtributeTrace=contact concretAtributeTrace = email concretAtributeClassTrace = Telephone }	

Tabela 24: Adição/Transformação de novos atributos

Em ambos casos, a tabela de rastreabilidade é essencial para verificar o refinamento do atributo do modelo.

Propriedade:

$$\forall abs, con \in Model.classesModel, traceAbs, traceCon \in Trace.allAtribute: (abs.nameAtribute - traceAbs.atributeAbs) \cup traceCon.atributeCon) \subseteq con.nameAtribute$$

Código Alloy:

assert NewAtribute{} *– ver anexo A.5*

Regra2b – Alteração das propriedades do atributo

Objectivo:

Identificar a existência de alterações de propriedades nos atributos no modelo destino em relação ao modelo original.

Descrição:

Verificar se as propriedades estruturais dos atributos do modelo original são iguais após o seu mapeamento no modelo destino.

Relativamente aos atributos e às operações, a alteração da visibilidade tem implicações mais profundas pois pode interferir directamente na forma como estes são acedidos.

Podemos considerar uma boa pratica de programação, que os atributos de uma classe sejam privados e possam ser acedidos através dos métodos por ela disponibilizados, tornando assim possível que as outras classes adjacentes os possam aceder.

Esta verificação é feita aos seguintes elementos:

- Visibilidade;
- Tipo; e
- Identificador.

Propriedade:

$$\forall abs, con \in Model.classesModel.atributesClass:$$
$$(abs.visibilityAtribute \neq con.visibilityAtribute) \vee$$
$$(abs.idAtribute = trace.atributeAbs \wedge con.idAtribute = trace.atributeCon) \wedge$$
$$(abs.nameAtribute \neq con.nameAtribute) \vee (abs.typeAtribute \neq con.typeAtribute)$$

Exemplo:

Modelo Original	Modelo Destino
	
Alloy	
sig nameA extends Atribute {}{ visibilityAtribute=publico nameAtribute=name typeAtribute=tipoSTR }	sig nameC extends Atribute {}{ visibilityAtribute=privado nameAtribute=name typeAtribute=tipoSTR }

Tabela 25: Alteração das propriedades do atributo - visibilidade

Código Alloy:

assert ChangeProprietyAtribute{} *– ver anexo A.5*

Regra2c – Mover um atributo duma subclasse para uma superclasse

Objectivo:

Identificar a existência de atributos comuns de subclasses que tenham a mesma superclasse.

Descrição:

Esta regra vai verificar a existência de atributos comuns em todas as subclasses de uma determinada superclasse, no modelo destino.

Propriedade:

$$\forall abs, con \in Model.relationsModel :$$
$$(abs.\sup erClass = con.\sup erClass) \wedge$$
$$(abs.subClass = con.subClass) \wedge$$
$$(abs.subClass \subseteq abs.\sup erClass) \wedge$$
$$(con.subClass \subseteq con.\sup erClass) \wedge$$
$$(abs.subClass.atributes \subseteq con.\sup erClass.atribute) \wedge$$
$$(abs.subClass.atributes \not\subset con.subClass.atribute)$$

Exemplo:

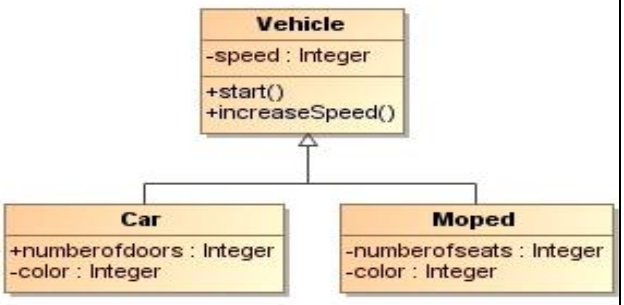
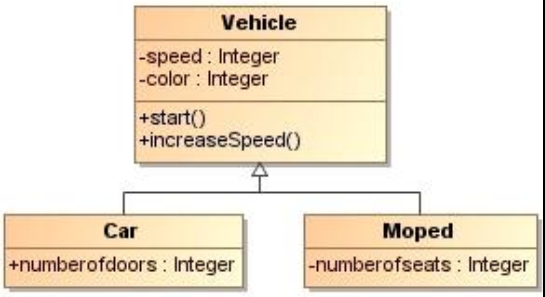
<table>
<tr><th>Modelo Original</th><th>Modelo Destino</th></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><th colspan="2">Alloy</th></tr>
<tr><td><pre>
sig nameA extends Atribute {}{
        visibilityAtribute=publico
        nameAtribute=name
        typeAtribute=tipoSTR   }
sig getNameA extends Operation{}{
        isAbstract=False
        visibilityOp=publico
        returnOp = Str
        nameOp=getName
        parameterOp=param   }
sig PersonA extends Class {}{
        isAbstract=False
        visibilityClass=publico
        nameClass=Person
        atributesClass = nameA
           operationsClass = getNameA}
</pre></td><td><pre>
sig nameC extends Atribute {}{
    visibilityAtribute=privado
        nameAtribute=name
        typeAtribute=tipoSTR }
sig getNameC extends Operation{}{
        isAbstract=False
        visibilityOp=publico
        returnOp = Str
        nameOp=getName
        parameterOp=param }
sig EmployeeC extends Class {}{
        isAbstract=False
        visibilityClass=publico
        nameClass=Employee
        atributesClass = nameC
        operationsClass = getNameC   }
</pre></td></tr>
<tr><td colspan="2">Rastreabilidade: Person->Employee</td></tr>
</table>

Tabela 26: Mover um atributo de subclasses para a sua superclasse

Código Alloy:

assert MovedAtributeGeneralization{} *– ver anexo A.5*

Regra2d – Mover um atributo para uma associação

Objectivo:

Identificar a existência de atributos do modelo original que após um refinamento estão numa classe adjacente.

Descrição:

Um cenário possível é um atributo de uma determinada classe passar para a sua adjacente. Neste contexto no modelo destino, o atributo pertence à classe adjacente.

Propriedade:

$$\forall abs,con \in Model.classesModel, trace \in Trace.allClass,$$
$$abs\,\mathrm{Re}\,l, con\,\mathrm{Re}\,l \in Model.relationsClass:$$
$$(abs.nameClass = trace.classAbs \wedge con.nameClass = trace.classCon) \vee$$
$$(abs.nameClass = con.nameClass) \wedge$$
$$(abs.atributesClass.nameAtribute = con.atributesClass.nameAtribute) \wedge$$
$$(abs.nameClass = con\,\mathrm{Re}\,l.start.class.nameClass) \wedge$$
$$(abs.nameClass = con\,\mathrm{Re}\,l.end.class.nameClass) \wedge$$
$$(con\,\mathrm{Re}\,l.relationTypeAss = agrega \vee con\,\mathrm{Re}\,l.relationTypeAss = associa)$$

Exemplo:

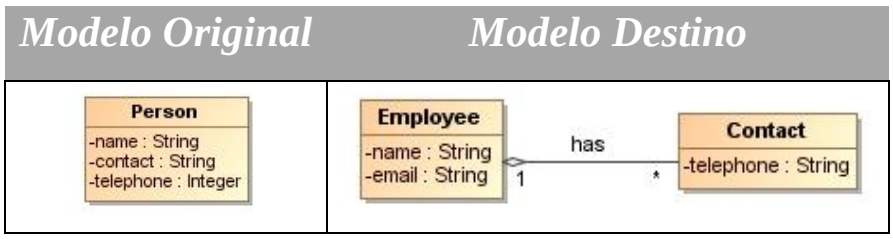


Tabela 27: Mover um atributo de uma classe para outra adjacente

Código Alloy:

assert MoveAtributeAssociation{} *– ver anexo A.5*

Regra2e – Transformar um atributo numa classe

Objectivo:

Identificar os atributos do modelo original que originaram novas classes no modelo destino.

Descrição:

Nesta regra a tabela de rastreabilidade é essencial. Pois é através das suas referências que temos o refinamento de atributos em classes.

Propriedade:

$$\forall abs, con \in Model.classesModel, trace \in Trace:$$
$$abs.idAtribute = trace.atributeAbs \wedge con.nameClass = trace.atributeToClassCon$$

Exemplo:

Mesmo da regra 2a – Exemplo Cenário 2

Código Alloy:

assert TurnAtributeIntoClass{} *– ver anexo A.5*

6.3.3 Regras sobre Métodos

Neste item, apresentamos as regras relativas à categoria de método. Na Tabela 28 temos o nome e uma descrição sintetizada das regras desta categoria.

Regra	Função	Descrição Síntese
Regra3a	Adição de Métodos.	Determinar os novos métodos no modelo destino.
Regra3b	Alteração das propriedades dos métodos.	Verificar as propriedades do método.
Regra3c	Alteração do número de parâmetros.	Verificação de métodos com o mesmo identificador mas com diferente número de parâmetros.

Tabela 28: Regras de refinamento de métodos

Regra3a – Adição de Métodos

Objectivo:

Identificar os novos métodos existentes no modelo destino, que não existem no modelo original.

Descrição:

Os métodos são as acções que uma classe pode realizar.

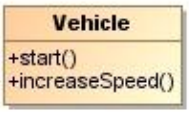
Esta regra devolve os novos métodos e respectivas classes existentes no modelo destino que não estejam identificadas no modelo original.

Consideramos ainda que um método que tem um identificador igual em ambos modelos, mas que apresenta um número diferente de parâmetros no modelo destino é um novo método.

Propriedade:

$$\forall abs, con \in Model.classesModel$$
$$abs.nameOperation \subseteq con.nameOperation$$

Exemplo:

Modelo Original	*Modelo Destino*
	

Alloy	

sig startA extends Operation{}{
isAbstract=False
visibilityOp=publico
returnOp = Void
nameOp=start
parameterOp.idPm=param
parameterOp.values.namePm=none
parameterOp.values.typePm=none}}
sig VehicleA extends Class {}{
isAbstract=False
visibilityClass=publico
nameClass=Person
atributesClass = none
operationsClass = startA}
	sig startC extends Operation{}{
	isAbstract=False
	visibilityOp=publico
	returnOp = Void
	nameOp=start
	parameterOp.idPm=param
	parameterOp.values.namePm=none
	parameterOp.values.typePm=none}
	sig increaseSpeedC extends Operation{}{
	isAbstract=False
	visibilityOp=publico
	returnOp = Void
	nameOp=increaseSpeed
	parameterOp.idPm=param
	parameterOp.values.namePm=none
	parameterOp.values.typePm=none}
	sig VehicleC extends Class {}{
	isAbstract=False
	visibilityClass=publico
	nameClass=Person
	atributesClass = none
	operationsClass=startC+increaseSpeedC}

Tabela 29: Adição de novos métodos

Código Alloy:

assert NewOperation{} *– ver anexo A.5*

Regra3b – Alteração das propriedades dos métodos

Objectivo:

Verificar se as propriedades estruturais dos métodos do modelo original são iguais no modelo destino.

Descrição:

Como as suas antecessoras, pretende-se verificar se as propriedades estruturais iniciais dos métodos do modelo original após o seu mapeamento são iguais no modelo destino.

Esta verificação é feita aos seguintes elementos:

- Identificador;
- Abstracção;
- Visibilidade;
- Número e tipo de parâmetros; e
- Resultado de retorno.

Propriedade:

$$\forall abs, con \in Model.classesModel.operationClass:$$
$$(abs..isAbstract \neq con.isAbstract) \vee$$
$$(abs.nameOp \neq con.nameOp) \vee$$
$$(abs.visibilityOp \neq con.visibilityOp) \vee$$
$$(abs.parameterOp \neq con.parameterOp) \vee$$
$$(abs.returnOp \neq con.returnOp)$$

Exemplo:

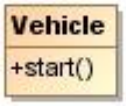
Modelo Original	*Modelo Destino*
	
Alloy	
`sig startA extends Operation{}{ isAbstract=False …… }`	`sig startC extends Operation{}{ isAbstract=True ….. }`

Tabela 30: Alteração das propriedades do método - visibilidade

Código Alloy:

assert ChangeProprietyOperation{} *– ver anexo A.5*

Regra3c – Mesmo nome diferente número de parâmetros

Objectivo:

Verificar a existência de métodos com o mesmo nome mas diferente número de parâmetros.

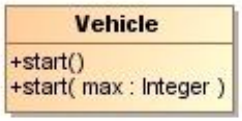
Descrição:

Consideramos que, em ambos modelos ou somente no modelo destino, um método pode ter o mesmo nome, mas diferente número de parâmetros. Nesta situação, estamos perante um novo método porque temos adição de nova informação.

Propriedade:

$$\forall abs, con \in Model.classesModel.operationClass :$$
$$(abs.nameOp = con.nameOp) \wedge (abs.parameterOp \neq con.parameterOp)$$

Exemplo:

Modelo Original	*Modelo Destino*
	
Alloy	

sig startA extends Operation{}{
isAbstract=False
visibilityOp=publico
returnOp = Void
nameOp=start
parameterOp.idPm=param
parameterOp.values.namePm=none
parameterOp.values.typePm=none}}
sig VehicleA extends Class {}{
isAbstract=False
visibilityClass=publico
nameClass=Person
atributesClass = none
operationsClass = startA}
	sig startC extends Operation{}{
	isAbstract= False
	visibilityOp=publico
	returnOp = Void
	nameOp=start
	parameterOp.idPm=param
	parameterOp.values.namePm=none
	parameterOp.values.typePm=none}
	sig start2C extends Operation{}{
	isAbstract= False
	visibilityOp=publico
	returnOp = Void
	nameOp=start
	parameterOp.idPm=param1
	parameterOp.values.namePm=maxC
	parameterOp.values.typePm=Integer}
	sig VehicleC extends Class {}{
	isAbstract=False
	visibilityClass=publico
	nameClass=Person
	atributesClass = none
	operationsClass=startC+ start2C }

Tabela 31: Operação com o mesmo nome mas diferente número de parâmetros

Código Alloy:

assert DiferentParameterOperation{} *– ver anexo A.5*

6.3.4 Regras sobre Relacionamentos

Neste item, começamos por apresentar as regras relativas à categoria de relacionamento. Na Tabela 32 temos uma descrição sintetizada das regras desta categoria.

Regra	Função	Descrição Síntese
Regra4a	Adição de Relacionamentos.	Determinar os novos relacionamentos existentes no modelo destino.
Regra4b	Promoção duma associação para agregação	Averiguar a existência de transformação do tipo associação para agregação.
Regra4c	Promoção duma associação para generalização	Averiguar a existência de transformação do tipo associação para generalização.
Regra4d	Promoção de relacionamentos muitos-para-muitos.	Averiguar a existência de associações do tipo muitos-para-muitos no modelo original e respectiva transformação no modelo destino.

Tabela 32: Regras de refinamento de relacionamentos

Regra4a – Adição de Relacionamentos

Objectivo:

Identificar os novos relacionamentos do modelo destino, que não estão no modelo original.

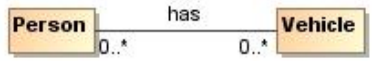
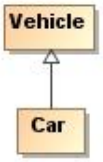
Descrição:

Esta regra devolve todos os novos relacionamentos e respectivas classes a si associadas.

Propriedade:

$$\forall abs, con \in Model.classesModel, abs\,\mathrm{Re}l, con\,\mathrm{Re}l \in Model.relationsModel,$$
$$traceAbs, traceCon \in Trace.allClass :$$
$$(abs.nameClass - traceAbs.classAbs) \cup traceCon.classCon \subseteq con.nameClass \wedge$$
$$(abs\,\mathrm{Re}l.relation \geq 0) \wedge (con\,\mathrm{Re}l.relation \geq 0)$$

Exemplo:

Modelo Original	*Modelo Destino*
	
Alloy	

`sig PersonVehicleAssA extends Association{}`
`{ visibilityAss=publico`
`nameAss=has`
`relationTypeAss=associa`
`start=PersonHasA`
`end=HasVehicleA }`
`sig PersonHasA extends AssociationPoint{}`
`{`
`class=PersonA`
`multiplicityAss=muitos`
`}`
`sig HasVehicleA extends AssociationPoint{}`
`{`
`class=VehicleA`
`multiplicityAss=muitos`
`}`
	`sig ArenaGameAssC extends Generalization {}`
	`{ visibilityGen=publico`
	`relationTypeGen=generaliza`
	`nameGen=has`
	`superClass=PersonA`
	`subClass=VehicleA`
	`}`

Tabela 33: Adição de novos relacionamentos

Código Alloy:

assert NewRelation{} *– ver anexo A.5*

Regra4b – Transformação de associação para agregação

Objectivo:

Identificar relacionamentos do tipo associação no modelo original que originem relacionamentos do tipo agregação.

Descrição:

Verificar a existência de relacionamentos do tipo associação no modelo original que após o seu mapeamento se transformam no tipo agregação, no modelo destino.

Propriedade:

$$\forall abs, con \in Model.relationModel :$$
$$(abs.namEAss = con.nameAss) \wedge$$
$$(abs.start.class = con.start.class) \wedge (abs.end.class = con.end.class) \wedge$$
$$(abs.start.multiplicity = con.start.multiplicity) \wedge (abs.end.multiplicity = con.end.multiplicity)$$
$$(abs.relationTypeAss = associa) \wedge (con.relationTypeAss = agrega)$$

Exemplo:

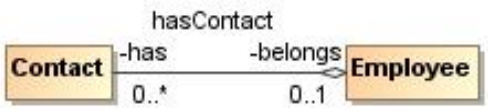
Modelo Original	*Modelo Destino*
	

Tabela 34: Transformação de uma associação numa agregação

Código Alloy:

assert AssociaToAgrega{} *– ver anexo A.5*

Regra4c – Transformação de associação para generalização

Objectivo:

Identificar relacionamentos com multiplicidade "um-para-um".

Descrição:

Verificar a existência de relacionamentos do tipo associação no modelo original que após o seu mapeamento se transformam no tipo generalização, no modelo destino.

Propriedade:

$$\forall abs, con \in Model.relationModel :$$
$$(abs.namEAss = con.nameGen) \wedge$$
$$abs.relationTypeAss = associa) \wedge$$
$$(con.relationTypeGen = generaliza) \wedge$$
$$(abs.start.multiplicity = um) \wedge$$
$$(abs.end.multiplicity = um) \wedge$$
$$(abs.start.class = con.end.class) \wedge$$
$$(abs.start.class = con.\text{sup}erClass)$$

Exemplo:

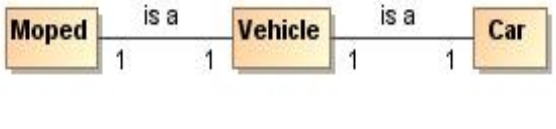
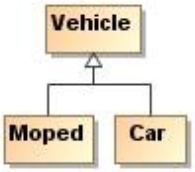
Modelo Original	*Modelo Destino*
	

Tabela 35: Transformação de uma associação numa generalização

Código Alloy:

assert AssociaToGeneraliza{} *– ver anexo A.5*

Regra4d – Transformação das associações do tipo "muitos-para-muitos"

Objectivo:

Identificar a existência de relacionamentos do tipo muito-para-muitos no modelo original e se estes são transformados no modelo destino.

Descrição:

Esta regra tem por intuito identificar a existência de relacionamentos de "muitos-para-muitos" do modelo original e se no modelo destino existe uma transformação que dá origem a dois novos relacionamento do tipo "um–para-muitos" e uma nova classe auxiliar denominada por classe de associação. Esta regra devolve as classes de associação e respectivos relacionamentos.

Propriedade:

$$\forall abs, con \in Model.relationModel :$$
$$(abs.start.multiplicity = muitos) \wedge (abs.end.multiplicity = muitos) \wedge$$
$$(con.start.class = con.end.class) \wedge$$
$$(con.start.multiplicity = muitos \wedge con.end.multiplicity = muitos)$$

Exemplo:

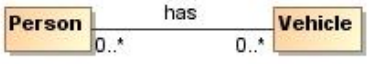
Modelo Original	*Modelo Destino*
	

Tabela 36: Transformação das associações do tipo "muitos-para-muitos"

Código Alloy:

assert ManyToMany {} *– ver anexo A.5*

6.4 Processo de Validação

No processo de validação, implementamos o algoritmo que nos permite deduzir a existência de refinamento.

Para esta avaliação podemos inferir dois tipos de refinamento distintos:

(i) Identidade – ambos os modelos (original e destino) possuem as mesmas propriedades a todos níveis, não existindo qualquer tipo de transformação, no entanto o simples processo de mapeamento é considerado um refinamento.

(ii) ConcretoRefinaAbstracto – o modelo destino apresenta um novo conjunto de elementos que não figuram no modelo original, cumprindo a semântica do modelo original.

Algoritmo: Transformações dedutivas

Sendo o objectivo deste trabalho verificar a existência de refinamento entre dois diagramas de classes, é necessário um método que infira se o modelo destino é um refinamento do modelo original.

Como dados de entrada temos o modelo original, o modelo destino e tabela de rastreabilidade.

Como dados de saída é produzido um conjunto de listas:

- regras de refinamento utilizadas – lista todos os identificadores de todas as regras utilizadas;
- novas classes – lista a novas classes do modelo destino;
- novos atributos– lista todos os novos atributose e respectivas classes;
- novas operações– lista todas as novas operações e respectivas classes; e
- novos relacionamentos– lista todos os novos relacionamentos.

Em UML, o diagrama de actividades permite especificar um conjunto de acções a realizar. Assim, a especificação do algoritmo é feita através de um conjunto de diagramas de actividade, nomeadamente os apresentados nas Figuras 22 a 29.

Para a inferência do refinamento entre os dois modelos, deve ser preservado a semântica do modelo original.

Na Figura 21 apresentamos o diagrama da actividade "I*nicial*". Esta é responsável por verificar qual o estado dos diagramas, isto é, se ambos estão ou não vazios. Este diagrama de actividades é composto pelas seguintes acções:

- Analisar qual o estado dos diagramas que servirão de entrada ao algoritmo;
- Analisar todas classes e relações.

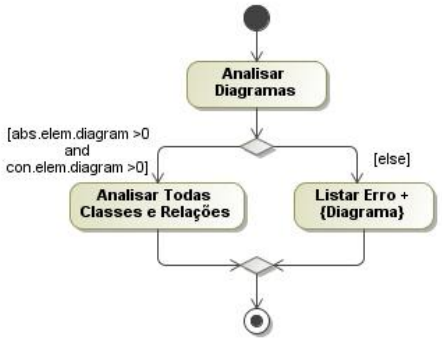


Figura 21: Diagrama de Actividade – Inicial

Na Figura 22 está apresentado a actividade "*Analisar Todas as Classes e Relações*". Esta pode ser executada de forma concorrente, onde é feita uma verificação inicial a ambos diagramas, analisando se o número de classes existentes no modelo original é diferente do número de classes existente no modelo destino, e se as relações no modelo original e no modelo destino são superiores a zero.

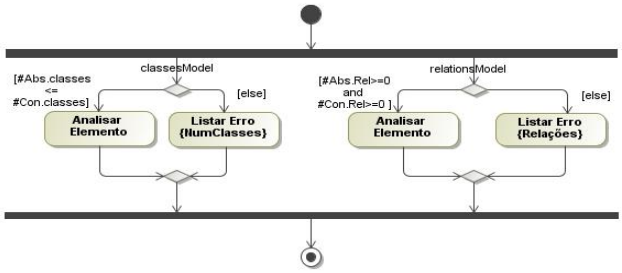


Figura 22: Diagrama de Actividade - Analisar Todas Classes e Relações

Na Figura 23 está apresentado o diagrama da actividade "*Analisar Elemento*". Nesta actividade, o objectivo é analisar concorrentemente as propriedades dos modelos verificando se as transformações realizadas mantêm a integridade dos dados. Assim, são executadas as actividades de "*Analisar Classe*", "*Analisar Atributo*", "*Analisar Operação*", "*Analisar Relacionamento*". Após a execução destas actividades é executada a actividade "*Apresentar Resultados*".

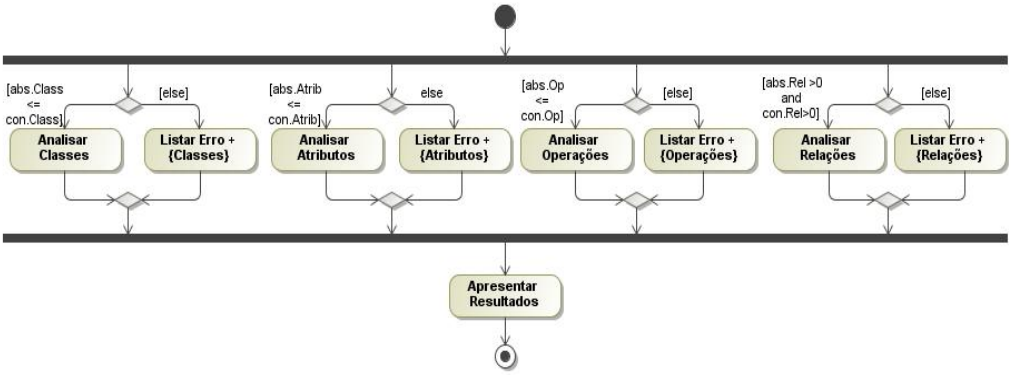


Figura 23: Diagrama de Actividade - Analisar Elemento

Na Figura 24 está representado o diagrama da actividade "Analisar Classes". Nesta actividade, começamos por inicializar a lista de regras de classe utilizadas – *ListClassRule*.

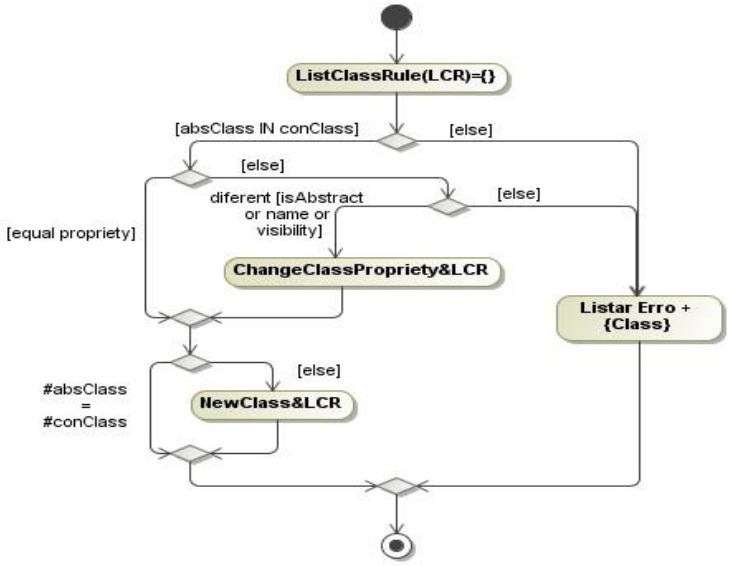


Figura 24: Diagrama de Actividade - Analisar Classes

Começamos por averiguar se as classes do modelo original estão contidas no modelo destino. Caso não estejam, é gerada a actividade *Listar Erro*, referente às Classes que significa erro. Caso contrário, procedemos à análise das regras de refinamento utilizadas.

De seguida, analisamos as propriedades das classes, caso estas não sejam iguais em ambos modelos então temos a regra "*ChangeProprietyClass*" sendo adicionado o seu identificador à *ListClassRule*, caso contrário procedemos à verificação da existência de novas classes, sem entrada *ListClassRule*.

Se o número de classes do modelo destino for superior às do modelo original então temos a regra "*NewClass*", e o mesmo procedimento anterior relativamente à *ListClassRule*. Caso contrário, concluímos a actividade - Analisar Classe.

Na Figura 25 está representado o diagrama da actividade "Analisar Atributos". Nesta actividade, começamos por inicializar a lista de regras de atributos utilizadas – *ListAtributeRule*.

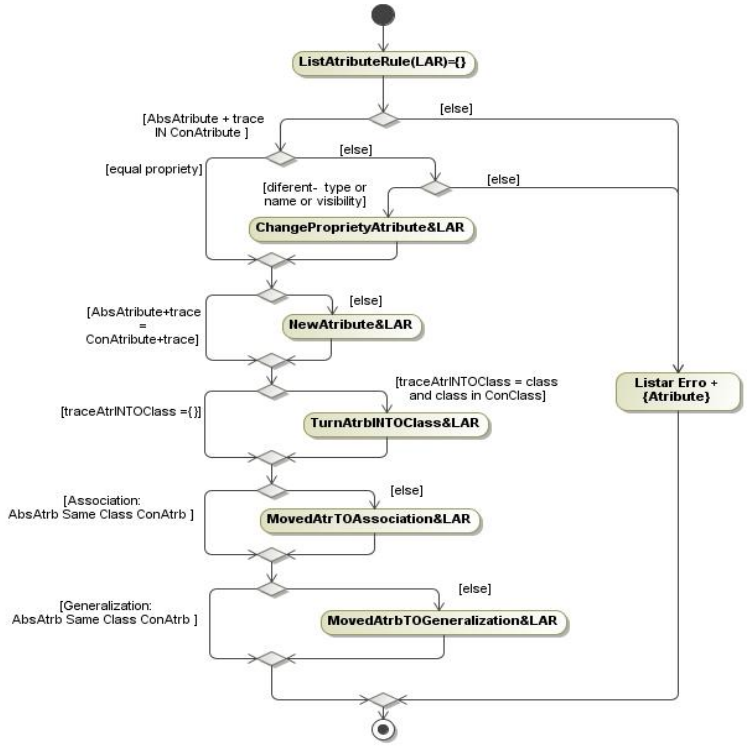


Figura 25: Diagrama de Actividade - Analisar Atributos

Começamos por averiguar se os atributos do modelo original estão contidos no modelo destino. Em caso não afirmativo, é gerada a actividade *Listar Erro*, referente aos Atributos, que significa erro. Caso contrário, procedemos à análise das regras de refinamento utilizadas.

De seguida, analisamos as propriedades dos atributos, caso estas não sejam iguais em ambos modelos então temos a regra "*ChangeProprietyAtribute*" sendo adicionado o seu identificador à *ListAtributeRule*, caso contrário procedemos à verificação da existência de novas classes, sem entrada na *ListAtributeRule*.

Se o número de atributos do modelo destino for superior às do modelo original então temos a regra "*NewAtribute*", e o mesmo procedimento anterior relativamente *ListAtributeRule*.

No caso de existir uma transformação de um atributo numa classe temos a regra "Turn*AtrIntoClass*", e o mesmo procedimento anterior relativamente *ListAtributeRule*.

No modelo original pudemos ter um atributo de uma classe que no modelo destino se apresente numa classe adjacente. Isto significa que existiu uma transformação do atributo através de uma associação ou numa generalização. Consoante for o caso

aplicamos as regras "*MovedAtrToAssociation*" ou "*MovedAtrToGeneralization*". Caso contrário, concluímos a actividade - Analisar Atributos.

Na Figura 26 está representado a actividade "Analisar Operações".

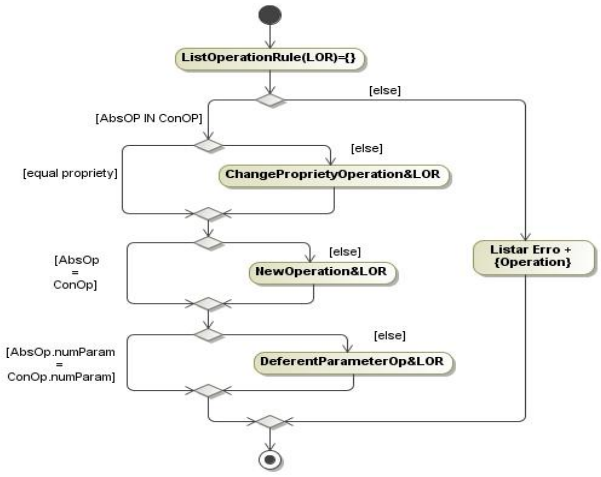


Figura 26: Diagrama de Actividade - Analisar Operações

Semelhante às suas antecessoras, temos a lista *ListOperationRule.*

Também, as regras "*ChangeProprietyOperation*" e "*NewOperation*" têm o mesmo enquadramento e procedimento das anteriores.

No caso de termos uma operação com o mesmo nome mas com número de parâmetros diferente, temos a regra "*DiferentParameterOp*".

Na Figura 27 está representado a actividade "Analisar Relacionamentos".

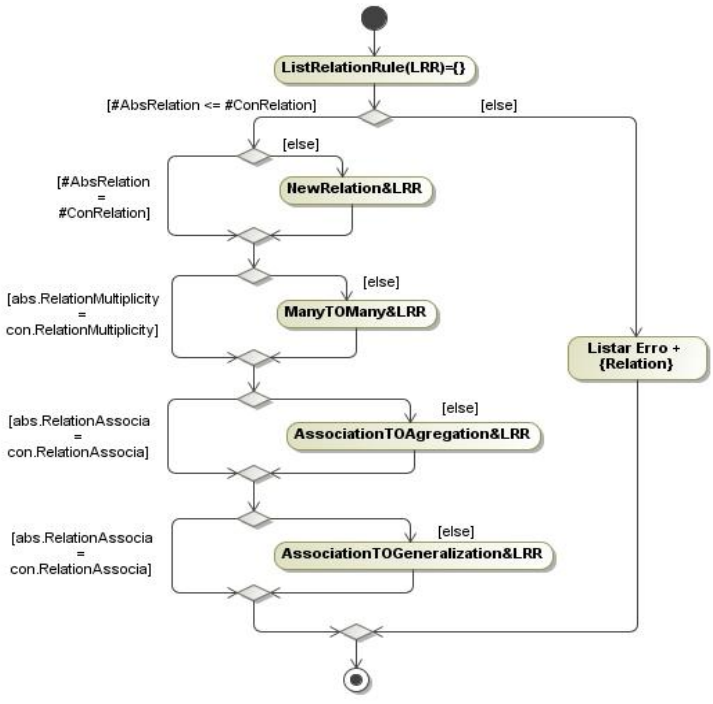


Figura 27: Diagrama de Actividade - Analisar Relacionamentos

Temos a inicialização da lista *ListRelationRule.*

Como já mencionado, consoante for o caso temos as regras "*NewRelation*", "*ManyToMany*", "*AssociationToAgregation*", "*AssociationToGeneralization*", sendo o tratamento destas regras idêntico às anteriores das outras categorias.

Na Figura 28 está representado a actividade "Apresentar Resultados". Nesta actividade, constrói-se uma lista global com todas as regras utilizadas e faz-se a respectiva visualização.

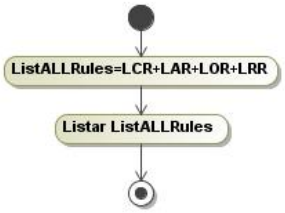


Figura 28: Diagrama de Actividade - Apresentar Resultados

6.5 Conclusão

O processo de transformação centraliza-se na forma de *como* devem ser mapeados os elementos do modelo original para os elementos de um modelo final.

O refinemento é composto por um conjunto de regras, cuja sua aplicação permite-nos deduzir as transformações existentes entre os modelos.

Apresentamos as regras de refinamento segundo 4 categorias distintas de elementos construtores de entidades: regras de Classe, regras de Atributo, regras de Operação e regras de Relacionamento.

Para deduzir a existência de refinamento criamos o algoritmo sistema de regras de refinamento representando as diferentes etapas em diferentes diagramas de actividades.

A actividade *Listar Erro* não foi possível implementar em Alloy devido às suas limitações de espaço.

7 Caso de Estudo

Em termos conceptuais, um "caso de estudo" pode-se definir como um conjunto de características associadas a um processo de recolha de dados e às estratégias de análise utilizadas.

Na secção 7.1 analisamos o conceito de caso de estudo e a sua relevância para este trabalho; na secção 7.2 apresentamos o caso de estudo Arena, apresentando um refinamento válido e outro inválido; e na secção 7.3 finalizamos este capítulo.

7.1 Motivação

O caso de estudo é uma metodologia de investigação do tipo empírica, adequada a trabalhos onde se procura compreender e analisar a aplicação de princípios que avaliem o impacto de determinados acontecimentos, em contextos reais.

Segundo alguns autores [47,48,49], o "caso" deve ser "único, específico, diferente, complexo", onde "as fronteiras entre o fenómeno e o contexto não estão bem definidos" [47], com "uma capacidade holística de preservar e compreender a totalidade e unicidade do caso".

Ainda, alguns autores advogam que o "caso de estudo" não constitui por si só uma metodologia de investigação específica, mas sim e apenas uma estratégia ou um processo técnico de especificação de um trabalho de investigação [48].

De forma sintetizada, podemos dizer que "o estudo de caso é uma investigação empírica que se baseia no raciocínio indutivo, dependente do trabalho de campo, não experimental, apoiado em diferentes fontes, técnicas e ferramentas de informação, deparando-se com o paradigma do '*porquê*' e do '*como*', para a correcta análise e compreensão da dinâmica do sistema".

Não é uma das finalidades de um caso de estudo gerar questões, no entanto quando estas surgem, ajudam a compreendê-lo e a analisá-lo melhor, assim como por vezes, conseguem promover objectivos que direccionam o nosso trabalho. Nesta óptica, propomos um caso de estudo, simplificado e restringido do sistema Arena.

7.2 Apresentação do Caso de Estudo Arena

O caso de estudo deve permitir analisar as propriedades das classes e das suas relações, assim como verificar a equivalência dos modelos.

O sistema Arena é um caso de estudo [38] cujo seu intuito é ilustrar o desenvolvimento de software orientado a objectos.

O ARENA é um sistema multi-utilizador, distribuído que permite a organização e orientação de torneios de jogos on-line. Este sistema permite que os organizadores se adaptem a novos jogos através de uma interface do jogo, e que anunciem e realizarem torneios com jogadores e espectadores via Web.

Como intervenientes temos: os organizadores (advertiser), a liga (leagueOwner), os utilizadores, os patrocinadores (sponsers), os jogos (game) e o torneio (tournament).

Entre as outras funções dos organizadores, estes podem definir novos estilos de torneios (diversificando e satisfazendo as expectativas dos seus jogadores e afins), descrevendo como os jogadores são inscritos nos jogos, monitorizando o ranking geral dos jogadores.

No nosso trabalho utilizaremos uma versão parcial do caso de estudo que nos permita verificar a implementação das regras de refinamento propostas.

Para proceder à análise do nosso caso de estudo, estruturaremos o processo com os seguintes passos:

- Identificação das entidades e relações que caracterizam o sistema Arena simplificado;
- Verificação da aplicação do sistema de regras de refinamento.

Ao nível de requisitos funcionais, na Tabela 37 apresentamos a descrição das entidades, atributos e relações que caracterizam o sistema proposto:

Entidade	***Descrição***	***Atributos&Relações***
Arena	Entidade principal do sistema.	– Max de torneios; – Leagues; – Grupos de Interesse.
Advertiser	Entidade que apresenta um banner de divulgação durante os jogos.	– Nome; – Account; – League.
LeagueOwner	Entidade que representa uma comunidade executora de torneios.	
User	Entidade que representa uma comunidade	– Nome

	que joga nos torneios.	
Game	Entidade que representa a competição entre os diferentes jogadores, segundo um conjunto de regras previamente estabelecidas.	
Tournament	Conjunto de jogos entre 2 ou mais jogadores. O torneio termina quando existir um único jogador.	– Nome; – Data Inicial Inscrição; – Data Final Inscrição; – Data Inicial Jogo; – Data Final Jogo; – Max de jogadores.

Tabela 37: Caso de estudo Arena - entidades e associações

7.2.1 Refinamento Válido

Segundo [29] em Alloy, o tempo de análise de um modelo com um número considerável de assinaturas pode ser demorado. O autor recomenda que a análise deve ser limitada a um âmbito de 5–10 assinaturas.

Na Tabela 38 apresentamos o modelo original e o modelo destino após uma etapa de refinamento. Para garantir a equivalência entre os modelos, quando temos determinado tipo de transformações é necessário a existência da tabela de rastreabilidade. Na Tabela 39 apresentamos a rastreabilidade onde temos caracterizado dois cenários possíveis: a transformação de atributos e a transformação de atributos em classes.

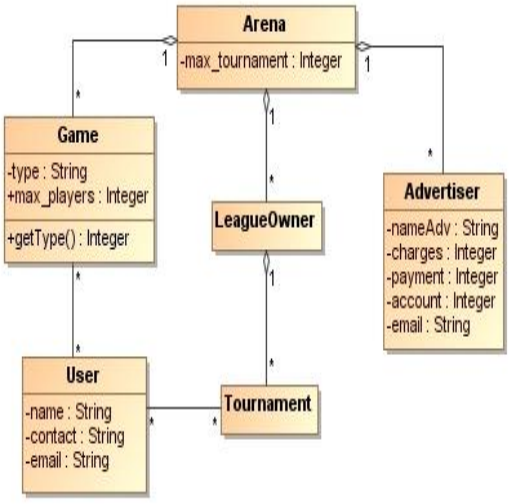
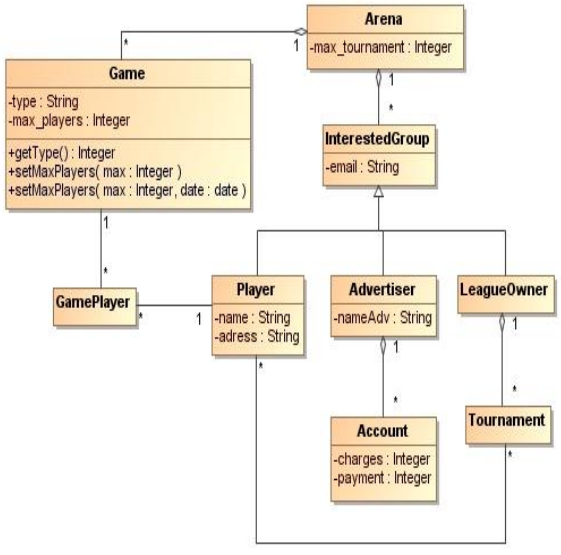
Modelo Original	Modelo Destino
	

Tabela 38: Caso de estudo Arena – modelo original e modelo destino

Class: User → Player
Atribute → Class: account → Account(charges+payment)

Tabela 39: Caso de estudo Arena - rastreabilidade

Podemos constatar que o modelo original é constituído por 6 classes, 6 relações e 13 elementos perfazendo um domínio de 25 elementos. Submetendo o modelo original a um refinamento, o modelo destino passa a ser constituído por 9 classes, 9 relações e 12 elementos, perfazendo no total um domínio de 30 elementos. Na Tabela 40 apresentamos o número de elementos por modelos.

	Modelo Original	Modelo Destino
# Classes	6	9
# Associações	6	6
# Generalizações	0	3
# Elementos	13	12

Tabela 40: Características dos Modelos

Submetendo os modelos ao nosso sistema de verificação, obtemos as seguintes regras:

- Relativamente à categoria Classe:
  - ***Adição de Classes*** **–** devolve as seguintes classes: *InterestedGroup*, *Account* e *Player*.
    - A classe *InterestedGroup* surge de uma opção de decisão para favorecer a organização do sistema de informação.
    - A classe *Account* surge do refinamento de um atributo do modelo original transformado numa classe no modelo destino.
    - A classe *Player* surge da necessidade de renomear a classe *User*. Esta alteração é fornecida pela tabela de rastreabilidade que identifica o nome da classe do modelo original, a ser renomeada no modelo destino.
  - ***Alteração das propriedades de uma classe*** **–** devolve a classe *Player* que surge da renomeação da classe *User*.
- Relativamente à categoria Atributo:
  - ***Adição de Atributos*** **–** devolve o atributo *address*.
  - ***Alteração das propriedades do atributo*** **–** no modelo original a propriedade de visibilidade do atributo *max-players* é pública, enquanto

no modelo original existe uma transformação dessa propriedade para privada.

- ***Mover um atributo duma subclasse para uma superclasse*** – no exemplo apresentado não existe esta regra.
- ***Mover um atributo para uma associação*** – no exemplo apresentado não existe esta regra.
- ***Transformar um atributo numa classe –*** no modelo original, o atributo account da classe Advertiser é transformado na classe Account no modelo destino.

- Relativamente à categoria Método:
  - ***Adição de Métodos –*** no modelo destino a classe *Game* tem os métodos setMaxPlayer (max: Integer), setMaxPlayer (max: Integer, date: DATE) que não figuram no modelo original.
  - ***Alteração das propriedades dos métodos*** – no exemplo apresentado não existe esta regra.
  - ***Mesmo nome diferente número de parâmetros*** – no modelo destino os métodos setMaxPlayer (max: Integer), setMaxPlayer (max: Integer, date: DATE) tem o mesmo nome mas diferente número de parâmetros.
- Relativamente à categoria Relacionamento:
  - ***Adições de Relacionamentos –*** todas as generalizações do modelo destino são novas relações assim como temos mais duas associações novas.
  - ***Transformação de associação para agregação*** – no exemplo apresentado não existe esta regra.
  - ***Transformação de associação para generalização*** – no exemplo apresentado não existe esta regra.
  - ***Transformação das associações do tipo "muitos-para-muitos" –*** no modelo original a associação entre as classes *Game* e *User* e a *User* e *Tournament* são do tipo de muitos-para-muitos. No modelo destino, a associação das classes *Game* e *User* é decomposta em duas associações com a classe de associação GamePlayer.

7.2.2 Refinamento Inválido

Com o intuito de apresentar um contra-exemplo da asserção "*ClassAbsINCon*" (classes do modelo original tem de estar contidas no modelo destino), criamos um modelo original com a classe "*ArenaA*" com o nome de correspondência "*User*" e um modelo destino com a classe "*ArenaC*" com o nome de correspondência "*Arena*".

Ao executar a seguinte asserção:

```
assert ClassAbsINCon{
    all abs:absClassName, con:conClassName, trace:traceConcretClass| abs in con+trace
}
```

foi gerado um contra-exemplo, que prova que a asserção é falsa.

Na Figura 29 temos apresentação do exemplo corrigido.

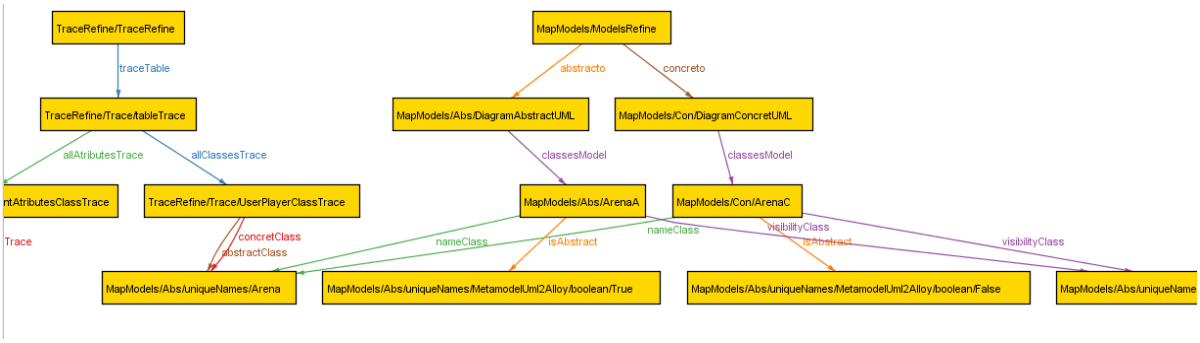


Figura 29: Exemplo correcto relativo à inclusão de classes

Na Figura 30 apresentamos o contra-exemplo gerado.

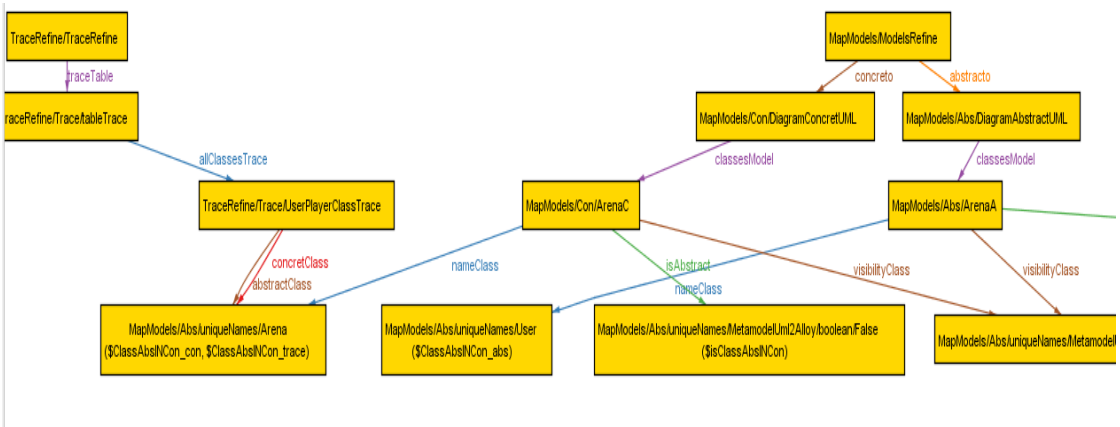


Figura 30: Contra-exemplo relativo à inclusão de classes

7.3 Conclusão

Neste capítulo apresentamos o caso de estudo Arena, com o objectivo de analisar as propriedades das classes e das suas relações, permitindo assim verificar a consistência e equivalência dos modelos.

Sendo o analisador do Alloy limitado no âmbito de pesquisa, apresentamos um sistema sintetizado, mas abrangente que permite a aplicação do sistema de regras de refinamento.

Apresentamos um exemplo de refinamento, segundo as regras propostas, e um refinamento inválido.

8 Conclusões e Trabalho Futuro

Como referido em capítulos anteriores, o desenvolvimento de software revela-se ser bastante consumidora de vários tipos de recursos. Em qualquer que seja o domínio da indústria exige-se, cada vez mais, capacidade de inovação e de adaptação, um bom desempenho, fiabilidade e robustez nos produtos. A evolução é uma consequência natural e ambicionável. Neste âmbito, a técnica de refinamento é um dos mecanismos a utilizar.

A linguagem UML é uma das linguagens de modelação mais utilizadas, pelos profissionais da indústria de software, para a criação de modelos. Na sua forma genérica, o UML apresenta ambiguidades o que pode levar a análises incorrectas dos modelos. Para além disso, o UML não tem suporte automático de análise de refinamentos.

Para colmatar a falta de formalismo do UML recorremos à linguagem formal Alloy. A linguagem de modelação Alloy fornece meios de especificação de modelos que podem ser analisadas automaticamente.

Na conjuntura deste trabalho, pensamos que UML juntamente com Alloy pode providenciar um consórcio vantajoso, isto porque ambas são compatíveis entre si e é fácil e intuitivo a escrita das restrições em Alloy, reforçando ainda a capacidade do Alloy oferecer um analisador automático.

A escolha da linguagem formal Alloy deve-se a alguns critérios, tais como, a representação intuitiva das construções das classes e a verificação automática de modelos. Além disso, esta linguagem é orientada a objectos, do tipo declarativo permitindo expressar restrições estruturais complexas, o que facilita o mapeamento dos diagramas de classes em modelos Alloy. Neste trabalho utilizamos o Alloy, e a sua ferramenta de análise automática para analisar a consistência do refinamento realizado.

As regras de refinamento garantem a análise e verificação automática da validade de um refinamento. Estas vão permitir verificar algumas propriedades algébricas do modelo original e do modelo destino.

Uma das limitações com que fomos confrontados, foi com o facto do Alloy apresentar uma performance baixa quando é necessário realizar análise de modelos com muitas assinaturas (segundo o caso de estudo temos mais de 40 assinaturas). O recomendado é uma análise limitada a um âmbito de 5–10 assinaturas. De forma a colmatar esta limitação, foi necessário reduzir o número de meta-classes do caso de estudo a serem representadas em Alloy.

8.1 Contribuições

O refinamento de um modelo pode introduzir problemas ao nível da preservação semântica. Neste âmbito foi construído um conjunto de regras formais cujo intuito é verificar a equivalência entre o modelo original e o modelo refinado, garantindo assim a preservação semântica entre os dois modelos.

Neste trabalho, definimos um mapeamento de um subconjunto do metamodelo UML para a linguagem Alloy. Este mapeamento permite representar e analisar os diagramas de classes em Alloy.

Foi também construído um sistema de regras de refinamento com objectivo de verificar algumas propriedades algébricas de um modelo original e do modelo destino através da análise automática usando o Alloy Analyzer. Este sistema de regras está organizado em várias categorias, segundo o seu tipo de construtor: classe, atributo, operação ou relacionamento.

Numa fase de análise ou de desenho do desenvolvimento do Software, consideramos que a implementação de uma metodologia de verificação de refinamento será uma mais-valia considerável, porque permite detectar erros subtis e inconsistências que podem ocorrer da reestruturação dos modelos. Esta metodologia fomenta:

- a diminuição de utilização de determinados recursos, aumentando o rigor e a fiabilidade das transformações realizadas, ampliando assim o grau de confiança;
- a adjacência do modelo refinado ao modelo de implementação, tornando o modelo refinado o mais próximo possível do modelo de implementação.

A implementação desta tecnologia seria muito util aos analistas e programadores, em todos os domínios da indústria.

8.2 Trabalho Futuro

Como trabalho futuro podemos sugerir um aprofundamento do tema, estendendo a capacidade de transformação incluindo mais elementos do UML, criar uma metodologia de trabalho e a construção de uma ferramenta capaz de traduzir automaticamente os diagramas de classes UML para modelos Alloy.

Ao nível da capacidade de extensão de transformação, um sistema que permita a possibilidade de verificação de refinamentos entre as diferentes etapas de desenvolvimento assim como adicionar novas regras ao sistema de refinamento.

Bibliografia

[1] Jacobson, I.; Booch, G.; Rumbaugh, J. E.: "The Unified Modeling Language Specification – User Guide", Addison-Wesley Publishing Company, 1999.

[2] Jackson, D.: Alloy Analyzer website http://alloy.mit.edu/.

[3] Hnatkowska B.; Huzar Z.; Kuźniarz L.; Tuzinkiewicz, L.: "On Understanding of Refinement Relationship", pg. 7-18, 3nd Workshop on Consistency Problems in UML-based Software Development, Lisboa, October 2004.

[4] Liu Z.; Li X.; Liu J.; Jifeng, H.: "Consistency and Refinement of UML Models", pg. 19-35, 3nd Workshop on Consistency Problems in UML-based Software Development, Lisboa, 2004.

[5] Straeten, R. V. D.; Simmonds, J.; Mens, T.: "Detecting Inconsistencies between UML Models Using Description Logics", Vrije Universiteit Brussel, System and Software Engineering Lab Brussels, Belgium, 2003.

[6] Massoni, T.; Gheyi, R.; Borba, P.:"Formal Refactoring for UML Class Diagrams", 19th Brazilian Symposium on Software Engineering, Brasil, 2005.

[7] Santos, A.; Menezes, L.; Cornélio, M.: "Flexible Transformation Language", pg. 9 – 14, WRT'07, University of Illinois at Urbana-Champaign, 2007.

[8] Evans, A. S.: "Reasoning with UML Class Diagrams", pg. 102-113, WIFT, 1998.

[9] Egyed, A.: "Consistent Adaptation and Evolution of Class Diagrams during Refinement", pg. 37-53, FASE, 2004.

[10] Jackson, D.; Wing, J.: "Lightweight formal methods," IEEE Computer, Apr. 1996.

[11] Netto, E. W.; Moreano, Nahri: "Método Z", IC-UNICAMP, Brasil, 2001.

[12] Nuka, G. website http://www.cs.kent.ac.uk/people/staff/gsn2/zeves/.

[13] Snook, C.; Butler, M.: "UML-B: Formal modeling and design aided by UML", pg. 92-122, ACM Trans. Softw. Eng. Methodol, 2006.

[14] Borges, R.M; Mota, A.C.: "Integrating UML and Formal Methods", pg. 97-112, Electr. Notes Theor. Comput. Sci. 184, 2007.

[15] OCL website http://www.omg.org/.

[16] Almeida, V.: "Uso da linguagem OCL no contexto de diagramas de classe da UML e programas em java", Universidade Federal de Minas Gerais, 2006.

[17] Bordbar, B.; Anastasakis, K.: "UML2Alloy: A Challenging Model Transformation", ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS – 2007), Nashville (TN), USA, 2007.

[18] Lano, K.; Bicarregui, J.: "UML Refinement and Abstraction Transformation", Second Workshop on Rigorous Object Orientated Methods, 1999.

[19] Massoni, T.; Gheyi, R.; Borba, P.: "A UML Class Diagram Analyzer", In 3rd International Workshop on Critical Systems Development with UML, affiliated with 7th UML Conference, Brasil, 2004.

[20] Goldstein, M.; Feldman, Y. A.; Tyszberowicz, S. S.: "Refactoring with Contracts", pg. 53-64. AGILE, 2006.

[21] Gamma, E.; Helm, R.; Johnson, R. E.; Vlissides, J.M.: "Design Patterns: Abstraction and Reuse of Object-Oriented Design". Pg. 406-431, ECOOP, 1993.

[22] Baruzzo, A.; Comini, M.: "Checking UML Model Consistency", University of Udine, Italia, 2006.

[23] Ammour, S.; Blanc, X.; Ziane, M.; Desfray, P.: "Improving Pattern Support in UML CASE Tools", pg. 107-116, 3nd Workshop on Consistency Problems in UML-based Software Development, Lisboa, October 2004.

[24] Richters, M.; Gogolla, M.: "On Formalization the UML Object Constraint Language OCL", pg. 449-464, Int. Conf. Conceptual Modeling (ER'98), 1998.

[25] Saaltink, M. : "The Z/EVES 2.0 User's Guide", Canada, 1999.

[26] Wirth N.: "Program Development by Stepwise Refinement", pp. 221-227, Communications of the ACM, Vol. 14, No. 4, 1971.

[27] Hoare, C.A.R.: "Communicating Sequential Processes", Prentice-Hall, 1985.

[28] Booch, G.; Jacobson, I.; Rumbaugh, J.: "UML Distelled", 2ª edição, Addison-Wesley, 2000.

[29] Daniel J.: "Software Abstractions: Logic, Language, and Analysis", London, England, 2006.

[30] Kalb, P.: "UML2Alloy: formalizing UML using Alloy and MDA technology", SS2008, Innsbruck, 2008.

[31] He, Y. : "Comparison of the Modeling Languages Alloy and UML", SERP 2006. Volume 2, Las Vegas, 2006.

[32] Vaziri, M.; Jackson, D.: "Some Shortcomings of OCL, the Object Constraint Language of UML", MIT Laboratory for Computer science, 1999.

[33] Borba, P.; Sampaio, A.; Cornélio, M.: "A Refinement Algebra for Object-Oriented Programming", 482 ECOOP-2003, Brasil, 2003.

[34] Baroni, L.; Brito e Abreu, F.: "Formalizing Object-Oriented Design Metrics upon the UML Meta-Model", École des Mines de Nantes, France Information Systems Group (INESC); FCT/Universidade Nova de Lisboa FCT/Universidade Nova de Lisboa, Brasil, 2002.

[35] Anastasakis, K; Bordbar, B: "UML2Alloy: A Tool For Lightweight Modelling Of Discrete Event Systems", pg. 209-216, IADIS International Conference in Applied Computing, Algarve, Portugal, 2005.

[36] Mostefaoui, F.; Vachon, J.: "Verification of Aspect-UML models using Alloy", pg. 41-48, Proceedings of the 10th international workshop on Aspect-oriented modeling, Vancouver, Canada, 2007.

[37] Jackson, D.: "Alloy 3.0 reference manual", May 2004.

[38] Bruegge, B.; Dutoit, A.H.: "Object-Oriented Software Engineering Using UML, Patterns, and Java", 2º edition, Prentice Hall, 2004.

[39] Silva, A.; Videira, C.: "UML: Metodologias e Ferramentas Case", 2ª edição, Centro Atlântico, Portugal, 2005.

[40] Baruzzo, A.: "UML Model Transformations for Quality and Correctness", Dipartimento di Matematica e Informatica, Università degli Studi di Udine, 2007, Itália.

[41] Briand, L. C.; Labiche, Y.; Yue, T.: "Automated traceability analysis for UML model refinements", Carleton University, Software Quality Engineering Laboratory, Ottawa, Canada, 2008.

[42] Software Quality Engineering Laboratory - http://squall.sce.carleton.ca/.

[43] D´ Souza, D.; Wills, A.: "Objects, Components and Frameworks with UML", Addison-Wesley, 1998.

[44] Pons, C.; Kutsche, R-D: "Traceability Across Refinement Steps in UML Modeling",3rd UML Workshop in Software Model Engineering (WiSME 2004) – UML 2004 Conference, 2004.

[45] Booch, G. "Object-Oriented Analysis and Design with Applications", 2ª edição, Addison-Wesley, 1995.

[46] Object Management Group - http://www.omg.org/.

[47] Yin, R.: "Case Stady Research:Design and Methods", 2ªEdiction, Sage Publication, 1994.

[48] Ponte, J.P.: "Estudo de caso em educação da matemática", Bolema, 2006.

[49] Punch, K.: “Introduction to Social Research: quantitative & qualitative approaches”, London: SAGE Publications, 1998.

Anexo A

A.1 Modelação do Metamodelo UML em Alloy

```
module RefinamentoDCAnaliseVerificacao/MetamodelUml2Alloy


open util/boolean
open RefinamentoDCAnaliseVerificacao/types

abstract sig Class {
	isAbstract: one Bool,
	visibilityClass: one idVisibility,
	nameClass: one idClass,
	atributesClass : set Atribute,
	operationsClass : set Operation	}
{
	isAbstract=False or isAbstract=True
	visibilityClass=publico or	visibilityClass=protegido or
	visibilityClass=privado
}
abstract sig Atribute {
	visibilityAtribute: one idVisibility,
	nameAtribute: one idAtribute,
	typeAtribute: one idTypeAtribute
}{
	visibilityAtribute=publico or	visibilityAtribute=protegido or
visibilityAtribute=privado
}
abstract sig Operation {
	isAbstractOp: one Bool,
	visibilityOp: one idVisibility,
	nameOp: one idOperation,
	parameterOp : one idParameter,
	returnOp : one idType
}{
	visibilityOp=publico or	visibilityOp=protegido or
visibilityOp=privado
}
abstract sig idParameter{
	idPm:one idName,
	values: set Parameter
}
abstract sig Parameter {
	namePm: set Atribute,
	typePm: one idTypeAtribute
}
abstract sig Association extends Relation {
	visibilityAss: one idVisibility,
	nameAss:one idName,
	relationTypeAss : one idRelation,
	start : one AssociationPoint,
	end : one AssociationPoint
}
```

```
{
	visibilityAss=publico or   visibilityAss=protegido or
	visibilityAss=privado or relationTypeAss=associa or
	relationTypeAss=agrega
}
abstract sig AssociationPoint extends Association {
	class: one Class,
	multiplicityAss: one idAssociationPoint
}
abstract sig Generalization extends Relation {
	relationTypeGen: one idRelation,
	nameGen:one idName,
	subClass: set Class,
	superClass: one Class
}{
	relationTypeGen=generaliza
}
abstract sig Model {
	classesModel : set Class,
	relationsModel : set Relation
}
```

A.2 Modelação da Rastreabilidade

```
module RefinamentoDCAnaliseVerificacao/MetaModelTraceability
open RefinamentoDCAnaliseVerificacao/MetaModelUml2Alloy

abstract  sig ModelTrace{
	allClassesTrace: set ClassesTrace,
	allAtributesTrace:set AtributesTrace
}
abstract  sig ClassesTrace extends ModelTrace{
	abstractClass: one idClass,
	concretClass: set idClass
}
abstract sig AtributesTrace extends ModelTrace{
	abstractAtributeTrace: one idAtribute,
	concretAtributeTrace: set idAtribute,
	concretAtributeClassTrace: set idClass
}
sig tableTrace extends ModelTrace{}{
	allClassesTrace= UserPlayerClassTrace
	allAtributesTrace= AdvertiserAccountAtributesClassTrace +
                                contactTOadressAtributesTrace
}
sig UserPlayerClassTrace extends ClassesTrace {}{
	abstractClass=User -- Arena
	concretClass=Player -- Arena
}
sig AdvertiserAccountAtributesClassTrace extends AtributesTrace{}{
	abstractAtributeTrace=account
	concretAtributeTrace = nenhum
	concretAtributeClassTrace =Account
}
sig contactTOadressAtributesTrace extends AtributesTrace{}{
	abstractAtributeTrace=contact
	concretAtributeTrace = adress
	concretAtributeClassTrace =none
}
 sig TraceRefine extends ModelTrace
{
	traceTable: one ModelTrace
}{
	traceTable =Trace/tableTrace
}
fun tableTrace:one ModelTrace{
	allTraceClassTable + allTraceAtributeTable
}
fun allTraceClassTable:one ModelTrace{
	traceTable[TraceRefine].allClassesTrace
}
fun allTraceAtributeTable:one ModelTrace{
	traceTable[TraceRefine].allAtributesTrace
}
fun allTraceClassAbsCon:set idClass {
	(traceAbstractClass + traceConcretClass + traceConcretAtributeClass)
}
fun traceAbstractClass:set idClass {
	tableTrace.abstractClass
}
```

```
fun traceConcretClass:set idClass {
      tableTrace.concretClass
}
fun traceConcretAtributeClass:set idClass {
      tableTrace.concretAtributeClassTrace
}
fun allTraceAtributesAbsCon:set idAtribute {
      (traceAbstractAtribute + traceConcretAtribute)
}
```

A.3 Modelação UniqueName, Type, MapModel

UniqueName

```
module RefinamentoDCAnaliseVerificacao/uniqueNames
open RefinamentoDCAnaliseVerificacao/MetamodelUml2Alloy

sig maxTournament, maxTournament1, type, maxPlayers, adress, nenhum extends
idAtribute{}
sig name, contact, email, charges, payment, account extends idAtribute{}
sig pm1, pm2 extends idParameter{}
sig Arena, Game, User, LeagueOwner, Tournament, Advertiser, Account,
ArenaGame extends idClass{}
sig InterestGroup, Player, GamePlayer, Match extends idClass{}
sig getMaxPlayer, setMaxPlayer extends idOperation{}
sig getMaxPlayerPm, has, hasPlayers, isPlayed, hasTournament, hasAdvertiser,
hasLeague, parm1, parm2  extends idName{}
sig setMaxPlayerPm, hasplayerLHS, hasplayerRHS, hasInterestGroup extends
idName{}
sig hasInterestGroupPlayer, hasInterestGroupAdvertiser,
hasInterestGroupLeagueOwner, advertiserAccount extends idName{}
sig NewClass, ChangeProprietyClass, IdentClass, CONRefinaABS,
identidadeRefina extends idRule{}
sig NewAtribute, ChangeProprietyAtribute, MovedAtributeAssociation,
MovedAtributeGeneralization, TurnAtributeINTOClass extends idRule{}
sig NewOperation, ChangeProprietyOperation, DiferentParameterOperation
extends idRule{}
sig NewRelation, AssociationTOAgregation, AssociationTOGeneralization,
MANYtoMANY  extends idRule{}
```

Type

```
module RefinamentoDCAnaliseVerificacao/types

abstract sig Type {}
abstract sig idType{}
abstract sig Relation {}
abstract sig idRelation{}

sig Void extends idType {}
sig Str extends idType {}
sig Integer extends idType {}
sig publico extends idVisibility{}
sig protegido extends idVisibility{}
sig privado extends idVisibility{}
sig associa extends idRelation{}
sig agrega extends idRelation{}
sig generaliza extends idRelation{}
sig zero extends idAssociationPoint{}
sig um extends idAssociationPoint{}
sig muitos extends idAssociationPoint{}
abstract sig idAssociationPoint{}
abstract sig idClass{}
abstract sig idName{}
abstract sig idAtribute {}
abstract sig idTypeAtribute{}
abstract sig idOperation{}
```

```
abstract sig idVisibility {}
abstract sig idRule {}
```

MapModel

```
module RefinamentoDCAnaliseVerificacao/MapModels

open RefinamentoDCAnaliseVerificacao/Models/DiagramAbstractUML as Abs
open RefinamentoDCAnaliseVerificacao/Models/DiagramConcretUML as Con
sig ModelsRefine extends Model {
      abstracto, concreto: one Model
}{
      abstracto=Abs/DiagramAbstractUML--LIM--Ident--Errado
    concreto=Con/DiagramConcretUML--LIM--Ident
}
fun absModel:set Model {
      abstracto[ModelsRefine]
}
fun absClass:set Class {
      abstracto[ModelsRefine].classesModel
}
fun absRelation:set Relation {
      abstracto[ModelsRefine].relationsModel
}
fun absClassName:set idClass {
      abstracto[ModelsRefine].classesModel.nameClass
}
fun absClassAtribute:set Atribute {
      abstracto[ModelsRefine].classesModel.atributesClass
}
fun absClassAtributeName:set idAtribute {
      abstracto[ModelsRefine].classesModel.atributesClass.nameAtribute
}
fun absClassAtributeType:set idType{
      abstracto[ModelsRefine].classesModel.atributesClass.typeAtribute
}
fun absClassOperation:set Operation {
      abstracto[ModelsRefine].classesModel.operationsClass
}
fun absClassOperationsName:set idOperation {
      abstracto[ModelsRefine].classesModel.operationsClass.nameOp
}
fun absRelationAssociationName:set idName {
      abstracto[ModelsRefine].relationsModel.nameAss
}
fun absRelationAssociationType:set idRelation {
      abstracto[ModelsRefine].relationsModel.relationTypeAss
}
fun absRelationAssociationStartClass:set Class {
      abstracto[ModelsRefine].relationsModel.start.class
}
fun absRelationAssociationEndClass:set Class {
      abstracto[ModelsRefine].relationsModel.end.class
}
fun absRelationAssociationStartMultiplicity:set idAssociationPoint  {
      abstracto[ModelsRefine].relationsModel.start.multiplicityAss
}
```

```
fun absRelationAssociationEndMultiplicity:set idAssociationPoint {
      abstracto[ModelsRefine].relationsModel.end. multiplicityAss
}
fun absRelationGeneralizationName:set idName {
      abstracto[ModelsRefine].relationsModel.nameGen
}
fun absRelationGeneralizationType:set idRelation {
      abstracto[ModelsRefine].relationsModel.relationTypeGen
}
fun absRelationGeneralizationSuperClass:set Class {
      abstracto[ModelsRefine].relationsModel.superClass
}
fun absRelationGeneralizationSubClass:set Class {
      abstracto[ModelsRefine].relationsModel.subClass
}
fun conModel:set Model {
      concreto[ModelsRefine]
}
fun conClass:set Class {
      concreto[ModelsRefine].classesModel
}
fun conRelation:set Relation {
      concreto[ModelsRefine].relationsModel
}
fun conClassName:set idClass {
      concreto[ModelsRefine].classesModel.nameClass
}
fun conClassAtribute:set Atribute {
      concreto[ModelsRefine].classesModel.atributesClass
}
fun conClassAtributeName:set idAtribute {
      concreto[ModelsRefine].classesModel.atributesClass.nameAtribute
}
fun conClassAtributeType:set idType{
      abstracto[ModelsRefine].classesModel.atributesClass.typeAtribute
}
fun conClassOperation:set Operation{
      concreto[ModelsRefine].classesModel.operationsClass
}
fun conClassOperationsName:set idOperation{
      concreto[ModelsRefine].classesModel.operationsClass.nameOp
}
fun conRelationAssociationName:set idName {
      concreto[ModelsRefine].relationsModel.nameAss
}
fun conRelationAssociationType:set idRelation {
      concreto[ModelsRefine].relationsModel.relationTypeAss     }
fun conRelationAssociationStartClass:set Class {
      concreto[ModelsRefine].relationsModel.start.class
}
fun conRelationAssociationEndClass:set Class {
      concreto[ModelsRefine].relationsModel.end.class
}
fun conRelationAssociationStartMultiplicity:set idAssociationPoint  {
      abstracto[ModelsRefine].relationsModel.start.multiplicityAss
}
fun conRelationAssociationEndMultiplicity:set idAssociationPoint {
      abstracto[ModelsRefine].relationsModel.end. multiplicityAss
}
```

```
fun conRelationGeneralizationName:set idName {
      concreto[ModelsRefine].relationsModel.nameGen
}
fun conRelationGeneralizationType:set idRelation {
      concreto[ModelsRefine].relationsModel.relationTypeGen
}
fun conRelationGeneralizationSuperClass:set Class {
      concreto[ModelsRefine].relationsModel.superClass
}
fun conRelationGeneralizationSubClass:set Class {
      concreto[ModelsRefine].relationsModel.subClass
}
fun idTONameClass(classMod: Model.classesModel, id:
Model.classesModel.nameClass): set Class{
      {classMod:classMod| some idClass:id|classMod.nameClass=idClass}
}
fun absClasses(modAbs, modCon:Model.classesModel): set Class {
      {clAbs:modAbs| some clCon:modCon| clAbs.nameClass =clCon.nameClass}
}
fun conClasses(modAbs, modCon:Model.classesModel): set Class {
      {clCon:modCon| some clAbs:modAbs| clAbs.nameClass =clCon.nameClass}
}
fun idClassTOClass(mod:Model.classesModel, idClss:idClass): set Class {
      {class:mod| some idCl:idClss| class.nameClass =idCl}
}
fun idAtributeTOAtribute(mod:Model.classesModel.atributesClass,
idClss:idAtribute): set Atribute {
      {class:mod| some idCl:idClss| class.nameAtribute =idCl}
}
fun idOperationTOperation(mod:Model.classesModel.operationsClass,
idClss:idOperation): set Operation{
      {class:mod| some idCl:idClss| class.nameOp =idCl}
}
fun idRelationAssTORelationAss(mod:Model.relationsModel, idClss:idName): set
Relation{
      {class:mod| some idCl:idClss| class.nameAss =idCl}
}
fun idRelationGenTORelationGen(mod:Model.relationsModel, idClss:idName): set
Relation{
      {class:mod| some idCl:idClss| class.nameGen =idCl}
}
```

A.4 Modelação dos Modelos

Abstracto

```
module RefinamentoDCAnaliseVerificacao/Models/DiagramAbstractUML
open RefinamentoDCAnaliseVerificacao/uniqueNames
sig maxTournamentA extends Atribute {}
{
    visibilityAtribute=privado
      nameAtribute=maxTournament
      typeAtribute=Str
}
sig ArenaA extends Class {}
{
      isAbstract=True--False--
      visibilityClass=privado --publico
      nameClass=Arena
      atributesClass = maxTournamentA
      operationsClass = none
}
sig typeA extends Atribute {}
{
    visibilityAtribute=privado
      nameAtribute=type
      typeAtribute=Str
}
sig maxPlayersA extends Atribute {}
{
    visibilityAtribute=publico
      nameAtribute=maxPlayers
      typeAtribute=Integer
}
sig getMaxPlayerA extends Operation{}
{
      isAbstractOp=True
      visibilityOp=privado
      returnOp = Integer --Str
      nameOp=getMaxPlayer
      parameterOp=param1A
}
sig param1A extends idParameter{}
{
      idPm=getMaxPlayerPm
      values=param1aA
}
sig param1aA extends Parameter {}
{
      namePm=nenhum
      typePm= Void
}
sig GameA extends Class {}
{
      isAbstract=False
      visibilityClass=publico
      nameClass=Game
      atributesClass = typeA+maxPlayersA
      operationsClass = getMaxPlayerA
}
```

```
sig ArenaGameAssA extends Association {}
{
      visibilityAss= publico
      nameAss=has
      relationTypeAss=associa --agrega
      start= hasGameA
      end= belongsA
}
sig hasGameA extends AssociationPoint{}
{
      class =ArenaA
      multiplicityAss =um
}
sig belongsA extends AssociationPoint{}
{
      class =GameA
      multiplicityAss =muitos
}
sig nameA extends Atribute {}
{
    visibilityAtribute=privado
      nameAtribute=name
      typeAtribute=Str
}
sig contactA extends Atribute {}
{
    visibilityAtribute=publico
      nameAtribute=contact
      typeAtribute=Integer
}
sig emailA extends Atribute {}
{
    visibilityAtribute=publico
      nameAtribute=email
      typeAtribute=Integer
}
sig UserA extends Class {}
{
      isAbstract=False
      visibilityClass=publico
      nameClass=User
      atributesClass = nameA + contactA+emailA
      operationsClass = none
}
sig GameUserManyTOManyA extends Association {}
{
      visibilityAss= publico
      nameAss=hasPlayers
      relationTypeAss=associa
      start= hasUserA
      end= playsGameA
}
sig hasUserA extends AssociationPoint{}
{
      class =GameA
      multiplicityAss =muitos
}
```

```
sig playsGameA extends AssociationPoint{}
{
	class =UserA
	multiplicityAss =muitos
}
sig LeagueOwnerA extends Class {}
{
	isAbstract=False
	visibilityClass=publico
	nameClass=LeagueOwner
	atributesClass = emailA
	operationsClass = none
}
sig TournamentA extends Class {}
{
	isAbstract=False
	visibilityClass=publico
	nameClass=Tournament
	atributesClass = none
	operationsClass = none
}
sig TournamentUserManyTOManyA extends Association {}
{
	visibilityAss= publico
	nameAss=isPlayed
	relationTypeAss=associa
	start= tourUserA
	end= userTourA
}
sig tourUserA extends AssociationPoint{}
{
	class =TournamentA
	multiplicityAss =muitos
}
sig userTourA extends AssociationPoint{}
{
	class =UserA
	multiplicityAss =muitos
}
sig LeagueOwnerTournamentAssA extends Association {}
{
	visibilityAss= publico
	nameAss=hasTournament
	relationTypeAss=agrega
	start=leagueTourA
	end=tourLeagueA
}
sig leagueTourA extends AssociationPoint{}
{
	class =LeagueOwnerA
	multiplicityAss =um
}
sig tourLeagueA extends AssociationPoint{}
{
	class =TournamentA
	multiplicityAss =muitos
}
```

```
sig chargesA extends Atribute {}
{
    visibilityAtribute=publico
      nameAtribute=charges
      typeAtribute=Integer
}
sig paymentA extends Atribute {}
{
    visibilityAtribute=publico
      nameAtribute=payment
      typeAtribute=Integer
}
sig accountA extends Atribute {}
{
    visibilityAtribute=publico
      nameAtribute=account
      typeAtribute=Integer
}
sig AdvertiserA extends Class {}
{
      isAbstract=False
      visibilityClass=publico
      nameClass=Advertiser
      atributesClass = emailA +nameA + chargesA + paymentA + accountA
      operationsClass = none
}
sig ArenaLeagueOwnerAssA extends Association {}
{
      visibilityAss= publico
      nameAss=hasLeague
      relationTypeAss=agrega
      start= hasLeagueA
      end= belongsArenaA
}
sig hasLeagueA extends AssociationPoint{}
{
      class =ArenaA
      multiplicityAss =um
}
sig belongsArenaA extends AssociationPoint{}
{
      class =LeagueOwnerA
      multiplicityAss =muitos
}
sig ArenaAdvertiserAssA extends Association {}
{
      visibilityAss= publico
      nameAss=hasAdvertiser
      relationTypeAss=agrega
      start= hasAdvertiserA
      end= belongsArenaAdvA
}
sig hasAdvertiserA extends AssociationPoint{}
{
      class =ArenaA
      multiplicityAss =um
}
```

```
sig belongsArenaAdvA extends AssociationPoint{}
{
	class =AdvertiserA
	multiplicityAss =muitos
}
sig DiagramAbstractUML extends Model{}
{
	classesModel =ArenaA +GameA + UserA +LeagueOwnerA + TournamentA
		+ AdvertiserA
	relationsModel = ArenaGameAssA + GameUserManyTOManyA +
			TournamentUserManyTOManyA  +  LeagueOwnerTournamentAssA  +
			ArenaLeagueOwnerAssA + ArenaAdvertiserAssA
}
```

Concreto

```
module RefinamentoDCAnaliseVerificacao/Models/DiagramConcretUML
open RefinamentoDCAnaliseVerificacao/uniqueNames
sig maxTournamentC extends Atribute {}
{
    visibilityAtribute=privado
      nameAtribute=maxTournament
      typeAtribute=Integer
}
sig ArenaC extends Class {}
{
	isAbstract=False
	visibilityClass=publico
	nameClass=Arena
	atributesClass = maxTournamentC
	operationsClass = none
}
sig typeC extends Atribute {}
{
    visibilityAtribute=privado
      nameAtribute=type
      typeAtribute=Str
}
sig maxPlayersC extends Atribute {}
{
    visibilityAtribute=privado
      nameAtribute=maxPlayers
      typeAtribute=Integer
}
sig getMaxPlayerC extends Operation{}
{
	isAbstractOp=False
	visibilityOp=publico
	returnOp = Integer --Str
	nameOp=getMaxPlayer
	parameterOp=param1C
}
sig param1C extends idParameter{}
{
	idPm=getMaxPlayerPm
	values=param1aC
}
```

```
sig param1aC extends Parameter {}
{
	namePm=nenhum
	typePm= Void
}
sig setMaxPlayerC extends Operation{}
{
	isAbstractOp=False
	visibilityOp=publico
	returnOp = Str
	nameOp=setMaxPlayer
	parameterOp=param2C
}
sig param2C extends idParameter{}
{
	idPm=setMaxPlayerPm
	values=param2aC
}
sig param2aC extends Parameter {}
{
	namePm=nenhum
	typePm= Void
}
sig setMaxPlayerC2 extends Operation{}
{
	isAbstractOp=False
	visibilityOp=publico
	returnOp = Str
	nameOp=setMaxPlayer
	parameterOp=param3C
}

sig param3C extends idParameter{}
{
	idPm=setMaxPlayerPm
	values=param3aC
}
sig param3aC extends Parameter {}
{
	namePm=type +maxPlayers
	typePm=Integer
}
sig GameC extends Class {}
{
	isAbstract=False
	visibilityClass=publico
	nameClass=Game
	atributesClass = typeC+maxPlayersC
	operationsClass = getMaxPlayerC+setMaxPlayerC+setMaxPlayerC2
}
sig ArenaGameAssC extends Association {}
{
	visibilityAss= publico
	nameAss=has
	relationTypeAss=agrega
	start= hasGameC
	end= belongsC
}
```

```
sig hasGameC extends AssociationPoint{}
{
      class =ArenaC
      multiplicityAss =um
}
sig belongsC extends AssociationPoint{}
{
      class =GameC
      multiplicityAss =muitos
}
sig nameC extends Atribute {}
{
    visibilityAtribute=privado
      nameAtribute=name
      typeAtribute=Str
}
sig adressC extends Atribute {}
{
    visibilityAtribute=publico
      nameAtribute=adress
      typeAtribute=Integer
}
sig emailC extends Atribute {}
{
    visibilityAtribute=publico
      nameAtribute=email
      typeAtribute=Integer
}
sig PlayerC extends Class {}
{
      isAbstract=False
      visibilityClass=publico
      nameClass=Player
      atributesClass = nameC + adressC
      operationsClass = none
}
sig GamePlayerC extends Class {}
{
      isAbstract=False
      visibilityClass=publico
      nameClass=GamePlayer
      atributesClass = none
      operationsClass = none
}
sig GamePlayerLHSAssC extends Association {}
{
      visibilityAss= publico
      nameAss=hasplayerLHS
      relationTypeAss=associa
      start= hasGamePlayerTOGameC
      end= playsGamePalyerTOGameC
}
sig hasGamePlayerTOGameC extends AssociationPoint{}
{
      class =GameC
      multiplicityAss =um
}
```

```
sig playsGamePalyerTOGameC extends AssociationPoint{}
{
	class =GamePlayerC
	multiplicityAss =muitos
}
sig GamePlayerRHSAssC extends Association {}
{
	visibilityAss= publico
	nameAss=hasplayerRHS
	relationTypeAss=associa
	start= playsGamePalyerTOPlayerC
	end= hasGamePlayerTOPlayerC
}
sig hasGamePlayerTOPlayerC extends AssociationPoint{}
{
	class =PlayerC
	multiplicityAss =um
}
sig playsGamePalyerTOPlayerC extends AssociationPoint{}
{
	class =GamePlayerC
	multiplicityAss =muitos
}
sig LeagueOwnerC extends Class {}
{
	isAbstract=False
	visibilityClass=publico
	nameClass=LeagueOwner
	atributesClass = none
	operationsClass = none
}
sig TournamentC extends Class {}
{
	isAbstract=False
	visibilityClass=publico
	nameClass=Tournament
	atributesClass = none
	operationsClass = none
}
sig TournamentPlayerManyTOManyC extends Association {}
{
	visibilityAss= publico
	nameAss=isPlayed
	relationTypeAss=associa
	start= tourPlayerC
	end= playerTourC
}
sig tourPlayerC extends AssociationPoint{}
{
	class =TournamentC
	multiplicityAss =muitos
}
sig playerTourC extends AssociationPoint{}
{
	class =PlayerC
	multiplicityAss =muitos
}
```

```
sig LeagueOwnerTournamentAssC extends Association {}
{
	visibilityAss= publico
	nameAss=hasTournament
	relationTypeAss=agrega
	start=leagueTourC
	end=tourLeagueC
}
sig leagueTourC extends AssociationPoint{}
{
	class =LeagueOwnerC
	multiplicityAss =um
}
sig tourLeagueC extends AssociationPoint{}
{
	class =TournamentC
	multiplicityAss =muitos
}
sig chargesC extends Atribute {}
{
	visibilityAtribute=publico
	nameAtribute=charges
	typeAtribute=Integer
}
sig paymentC extends Atribute {}
{
	visibilityAtribute=publico
	nameAtribute=payment
	typeAtribute=Integer
}
sig AdvertiserC extends Class {}
{
	isAbstract=False
	visibilityClass=publico
	nameClass=Advertiser
	atributesClass = nameC
	operationsClass = none
}
sig AccountC extends Class {}
{
	isAbstract=False
	visibilityClass=publico
	nameClass=Account
	atributesClass = chargesC + paymentC
	operationsClass = none
}
sig AdvertiserAccountAssC extends Association {}
{
	visibilityAss= publico
	nameAss=advertiserAccount
	relationTypeAss=agrega
	start= hasAdvertiserAccountC
	end= belongsAdvertiserAccountC
}
sig hasAdvertiserAccountC extends AssociationPoint{}
{
	class =AdvertiserC
	multiplicityAss =um		}
```

```
sig belongsAdvertiserAccountC extends AssociationPoint{}
{
	class =AccountC
	multiplicityAss =muitos
}
sig InterestGroupC extends Class {}
{
	isAbstract=False
	visibilityClass=publico
	nameClass=InterestGroup
	atributesClass = emailC
	operationsClass = none
}
sig ArenaInterestGroupAssC extends Association {}
{
	visibilityAss= publico
	nameAss=hasInterestGroup
	relationTypeAss=agrega
	start= hasInterestGroupC
	end= belongsInterestGroupC
}
sig hasInterestGroupC extends AssociationPoint{}
{
	class =ArenaC
	multiplicityAss =um
}
sig belongsInterestGroupC extends AssociationPoint{}
{
	class =InterestGroupC
	multiplicityAss =muitos
}
sig InterestGroupPlayerGenC extends Generalization{}
{
	relationTypeGen=generaliza
	nameGen=hasInterestGroupPlayer
	subClass=PlayerC
	superClass=InterestGroupC
}
sig InterestGroupAdvertiserGenC extends Generalization{}
{
	relationTypeGen=generaliza
	nameGen=hasInterestGroupAdvertiser
	subClass=AdvertiserC
	superClass=InterestGroupC
}
sig InterestGroupLeagueOwnerGenC extends Generalization{}
{
	relationTypeGen=generaliza
	nameGen=hasInterestGroupLeagueOwner
	subClass=LeagueOwnerC
	superClass=InterestGroupC
}
sig DiagramConcretUML extends Model{}
{
	classesModel =ArenaC +GameC + PlayerC +LeagueOwnerC+TournamentC
		+ AdvertiserC + AccountC + GamePlayerC + InterestGroupC
	relationsModel = ArenaGameAssC + TournamentPlayerManyTOManyC
		+LeagueOwnerTournamentAssC +ArenaInterestGroupAssC
```

```
                    +GamePlayerRHSAssC       +       GamePlayerLHSAssC       +
                    InterestGroupLeagueOwnerGenC   +InterestGroupPlayerGenC  +
                    InterestGroupAdvertiserGenC + AdvertiserAccountAssC
}
```

A.5 Modelação do Sistema de Regras de Refinamento

```
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule1/Rule1InclusionClass as r1
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule1/Rule1IsAbstractClass as r1a
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule1/Rule1ChangeNameClass as r1b
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule1/Rule1VisibilityClass as r1c
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule1/Rule1InclusionClass as r1d
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule2/Rule2InclusionAtribute as r2In
```

Categoria Classes

```
assert NewClass{
  all newCls:isNewClass[], rul:r1/isClassAbsINCon[]| newCls=True and rul=True
}
fun isNewClass: one Bool{
      #mapNewClass>0 => True else False
}
fun mapNewClass: set Class{
      idClassTOClass[conClass, (conClassName + allTraceClassAbsCon) –
      absClassName]
}
assert ChangeProprietyClass{
            all chg:isChangeProprietyClass[],
            rul:r1d/isClassAbsINCon[]|chg=True and rul=True
}
fun isChangeProprietyClass(): one Bool{
      ((r1a/isAbstractClass =True) or
      (r1b/isChangeNameClass=True) or
      (r1c/isVisibilityClass=True))
      => True else False
}
assert ClassAbsINCon{
            all absIncon:isClassAbsINCon[]| absIncon =True
}
fun isClassAbsINCon(): one Bool{
      absINcon[]   => True else False
}
pred absINcon(){
      absClassName  in (conClassName +allTraceClassAbsCon)
}
pred ChangeNameClass{
            isChangeNameClass [] = True
}
fun isChangeNameClass(): one Bool{
            #conSameIdDiferentNameClass >0  => True else False
}
fun mapClassSameIdDiferentName(): set idClass->idClass{
            absSameIdDiferentNameClass[]->conSameIdDiferentNameClass[]
}
fun absSameIdDiferentNameClass(): set idClass{
      ((conClass.nameClass+traceAbstractClass)-
      {con:conClass.nameClass|some abs:absClass.nameClass|abs=con})
            &traceAbstractClass
}
fun conSameIdDiferentNameClass(): set idClass{
      ((conClass.nameClass+traceAbstractClass)-
      {con:conClass.nameClass|some abs:absClass.nameClass|abs=con})
            &traceConcretClass
}
```

```
pred AbstractClass{
            all isAbst:isAbstractClass[]|isAbst=True
}
fun isAbstractClass(): one Bool {
             #diferentIsAbstractClass[] >0 =>True else False
}
fun mapIsAbstractClass(): set Class->Bool{
      diferentIsAbstractClassABS->diferentIsAbstractClassABS.isAbstract+
      diferentIsAbstractClassCON->diferentIsAbstractClassCON.isAbstract
}
fun diferentIsAbstractClassABS(): set Class{
      idClassTOClass[absClass, diferentIsAbstractClass[]]
}
fun diferentIsAbstractClassCON(): set Class{
      idClassTOClass[conClass, diferentIsAbstractClass[]]
}
fun diferentIsAbstractClass(): set idClass{
      {difAbs:idClass|some abs: absClass, con:conClass|
            (abs.nameClass=con.nameClass and difAbs=con.nameClass and
                  abs.isAbstract!=con.isAbstract)}
}
pred SameProprietyClass{
             isSameProprietyClass[]=True
}
fun isSameProprietyClass(): one Bool{
      (r1IsAbstract/isAbstractClass =True) and
      (r1SameNameClass/isSameNameClass = True) and
      (r1VisibilityClass/isVisibilityClass = True)
      => True else False
}
pred VisibilityClass{
             isVisibilityClass[]=True
}
fun isVisibilityClass(): one Bool{
             #mapDiferentVisibilityClass[] >0 => True else False
}
fun mapDiferentVisibilityClass(): set Class->idVisibility{
      diferentVisibilityClassABS->diferentVisibilityClassABS.visibilityClass
                                    +
      diferentVisibilityClassCON->diferentVisibilityClassCON.visibilityClass
}
fun diferentVisibilityClassABS(): set Class{
      idClassTOClass[absClass, diferentVisibilityProprietyClass[]]
}
fun diferentVisibilityClassCON(): set Class{
      idClassTOClass[conClass, diferentVisibilityProprietyClass[]]
}
fun diferentVisibilityProprietyClass(): set idClass{
      {aux:idClass| some abs:absClass, con:conClass
            | abs.visibilityClass!=con.visibilityClass and
            abs.nameClass=con.nameClass and aux=con.nameClass }
}
```

Categoria Atributos

```
assert AtributesAbsINCon{
      all atrbs:isAtributesAbsINCon[], rul:r1d/isClassAbsINCon[]|atrbs=True
      and rul=True
}
```

```
fun isAtributesAbsINCon(): one Bool{
      atribAbsINCon[]    => True else False
}
pred atribAbsINCon(){
      (absClassAtributeName - traceAbstractAtribute + traceConcretAtribute)
      in conClassAtributeName         }
assert ChangedProprietyAtribute{
      all chAtr:isChangeProprietyAtribute[], rul:r1/isClassAbsINCon[],
      rul2:r2In/isAtributesAbsINCon[]|chAtr=True and rul=True and rul2=True
}
fun isChangeProprietyAtribute:one Bool{
      (r2Type/isTypeAtributeChange=True or
      r2Vis/isVisibilityAtributeChange = True or
      r2DifName/isDiferentNameAtribute=True)=> True else False
}
pred  DiferentNameAtribute{
             isDiferentNameAtribute[]=True
}
fun isDiferentNameAtribute(): one Bool{
             #mapDiferentNameAtribute[] >0 => True else False
}
fun mapDiferentNameAtribute(): set Atribute->Atribute{
      idAtributeTOAtribute[absClassAtribute, auxDifNameAbs[]]->
      idAtributeTOAtribute[conClassAtribute, auxDifNameCon[]]
}
fun auxDifNameCon(): set idAtribute{
      {traceCon:AtributesTrace.concretAtributeTrace| traceCon!=nenhum}
}
fun auxDifNameAbs(): set idAtribute{
      {traceCon:AtributesTrace| some con:auxDifNameCon[]|
      traceCon.concretAtributeTrace= con}.abstractAtributeTrace
}
assert NewAtribute{
      all nwAtr:isNewAtribute[], rul:r1/isClassAbsINCon[],
      rul2:r2In/isAtributesAbsINCon[]|nwAtr=True and
      atributeTraceINConcret[]=True and rul=True and rul2=True
}
fun isNewAtribute: one Bool{
      #mapNewAtribute>0 => True else False
}
fun mapNewAtribute: set Atribute->Class{
      mapNewAtributeClass->{c:conClass|
                           some c.atributesClass&mapNewAtributeClass }
}
fun mapNewAtributeClass: set Atribute{
      idAtributeTOAtribute[ conClassAtribute ,conNewAtribute[]]
}
fun conNewAtribute: set idAtribute{
      (conClassAtributeName + allTraceAtributesAbsCon)- absClassAtributeName
}
fun atributeTraceINConcret:one Bool{
      (traceConcretAtribute in conClassAtributeName) => True else False
}
assert TurnAtributeINTOClass{
      all atrToCls:isTurnAtributeINTOClass[],
      rul:r1/isClassAbsINCon[],rul2:r2In/isAtributesAbsINCon[]|
      atrToCls=True and rul=True and rul2=True
}
```

```
fun isTurnAtributeINTOClass(): one Bool{
      (#mapAtributeTurnINTOClass>0)=> True else False
}
fun mapAtributeTurnINTOClass(): set Atribute->Class{
      {a:idAtributeTOAtribute[absClassAtribute,
      traceAtributeTurnINTOClass.abstractAtributeTrace],
      b:idClassTOClass[conClass,
      traceAtributeTurnINTOClass.concretAtributeClassTrace]|
      some c:conClass| b in c}
}
fun traceAtributeTurnINTOClass(): set ModelTrace {
      {a:allTraceAtributeTable| #(a.abstractAtributeTrace)>0 and
      #(a.concretAtributeClassTrace)>0 }
}
pred TypeAtributeChange{
      isTypeAtributeChange =True
}
fun isTypeAtributeChange (): one Bool{
      (#diferentTypeAtribute>0) =>True else False
}
fun mapDiferentTypeAtribute(): set Atribute->idType {
      diferentTypeAtributeABS->diferentTypeAtributeABS.typeAtribute +
      diferentTypeAtributeCON->diferentTypeAtributeCON.typeAtribute
}
fun diferentTypeAtribute(): set idAtribute{
      {atr:idAtribute|some c:conClassAtribute, a:absClassAtribute|
      a.typeAtribute!=c.typeAtribute and a.nameAtribute=c.nameAtribute
      and atr=a.nameAtribute and atr=c.nameAtribute }
}
fun diferentTypeAtributeABS(): set Atribute{
      idAtributeTOAtribute[absClassAtribute, diferentTypeAtribute[]]
}
fun diferentTypeAtributeCON(): set Atribute{
      idAtributeTOAtribute[conClassAtribute, diferentTypeAtribute[]]
}
pred VisibilityAtributeChange{
       isVisibilityAtributeChange[]=True
}
fun isVisibilityAtributeChange(): one Bool{
       #mapDiferentVisibilityAtribute[] >0 => True else False
}
fun mapDiferentVisibilityAtribute(): set Atribute->idVisibility{
      diferentVisibilityAtributeABS->
      diferentVisibilityAtributeABS.visibilityAtribute +
      diferentVisibilityAtributeCON->
      diferentVisibilityAtributeCON.visibilityAtribute
}
fun diferentVisibilityAtributeABS(): set Atribute{
      idAtributeTOAtribute[absClassAtribute,
      diferentVisibilityProprietyAtribute[]]
}
fun diferentVisibilityAtributeCON(): set Atribute{
      idAtributeTOAtribute[conClassAtribute,
      diferentVisibilityProprietyAtribute[]]    }
fun diferentVisibilityProprietyAtribute(): set idAtribute{
      {a:idAtribute|some abs:absClassAtribute, con:conClassAtribute
      | abs.visibilityAtribute!=con.visibilityAtribute and
      abs.nameAtribute=con.nameAtribute and
      a=abs.nameAtribute and a=con.nameAtribute}      }
```

```
assert MoveAtributeAssociation{
       all mvAtr:isMoveAtributeAssociation, rul:r1/isClassAbsINCon[],
       rul2:r2In/isAtributesAbsINCon[]|mvAtr=True and rul=True and rul2=True
}
fun isMoveAtributeAssociation(): one Bool{
        #mapMovedAtributeClass>0 => True else False
}
fun mapMovedAtributeClass(): set Class->Atribute{
       movedAtributeClass[absClass]->
       idAtributeTOAtribute[absClassAtribute,movedAtributeAssociation[]]+
       movedAtributeClass[conClass]->
       idAtributeTOAtribute[conClassAtribute,movedAtributeAssociation[]]
}
fun movedAtributeClass(mod:Model.classesModel): set Class{
       {cls:mod| some mvAtr:movedAtributeAssociation[]|
       mod.atributesClass.nameAtribute=mvAtr}
}
fun movedAtributeAssociation(): set idAtribute{
       {atr:idAtribute| some abs:absClass, con:conClass, conRel:conRelation,
       traceAbs:traceAbstractClass, traceCon:traceConcretClass|
       abs.nameClass=traceAbs and con.nameClass=traceCon and
       abs.nameClass=con.nameClass and
       abs.atributesClass.nameAtribute=con.atributesClass.nameAtribute and
       abs.nameClass=conRel.start.(class+end).nameClass and
       (conRel.relationTypeAss=associa or conRel.relationTypeAss=agrega) }
}
assert MovedAtributeGeneralization{
       all mapMv:mapMovSuperAtribute[], rul:r1/isClassAbsINCon[],
       rul2:r2In/isAtributesAbsINCon[]|#mapMv>=0  and rul=True and rul2=True
}
fun isMovedAtributeGeneralization: one Bool{
             (#mapMovSuperAtribute>0 ) =>True else False
}
fun moveAtributeGeneralization(): set Class{
       {cls:conClass| some cls.atributesClass}
}
fun paiClass(mod:Model.relationsModel):set Class{
       {pai:{filho:mod| some filho.subClass}.superClass}
}
fun filhoClass(mod:Model.relationsModel):set Class{
       {filho:{pai:mod| some pai.superClass}.subClass}
}
fun sameAbsSubSupClassCon: set idClass{
       { cls:idClass| some subAbs:filhoClass[absRelation].nameClass,
       subCon:filhoClass[conRelation].nameClass,
       supAbs:filhoClass[absRelation].nameClass,
       supCon:filhoClass[conRelation].nameClass| subAbs=subCon and
       supAbs=supCon and paiClass[conRelation] in {rel:Generalization|
       rel.subClass in filhoClass[conRelation]}.superClass and
       paiClass[absRelation] in {rel:Generalization| rel.subClass in
       filhoClass[absRelation]}.superClass and
       paiClass[conRelation].atributesClass.nameAtribute  in
       {rel:Generalization| rel.subClass in
       filhoClass[absRelation]}.subClass.atributesClass.nameAtribute and
       filhoClass[conRelation].atributesClass.nameAtribute not in
       {rel:Generalization| rel.subClass in
       filhoClass[absRelation]}.subClass.atributesClass.nameAtribute }
}
```

```
fun sameAtrbSubAbsSuperCon: set idAtribute{
      idClassTOClass[absClass,sameAbsSubSupClassCon[]].atributesClass.nameAt
ribute&paiClass[conRelation].atributesClass.nameAtribute
}

fun mapMovSuperAtribute: set Class->Atribute{
      paiClass[conRelation]->
idAtributeTOAtribute[conClassAtribute,sameAtrbSubAbsSuperCon[]]
}
```

Categoria Operações

```
pred IsAbstractOperation{
            isAbstractOperation[] =True
}
fun isAbstractOperation(): one Bool {
             #mapIsAbstractOperation[ ]>0 =>True else False
}
fun mapIsAbstractOperation(): set Operation->Bool{
      diferentIsAbstractOpABS->diferentIsAbstractOpABS.isAbstractOp
      +diferentIsAbstractOpCON->diferentIsAbstractOpCON.isAbstractOp
}
fun diferentIsAbstractOpABS(): set Operation{
      idOperationTOperation[absClassOperation, IsAbstractOperationDif[]]
}
fun diferentIsAbstractOpCON(): set Operation{
      idOperationTOperation[conClassOperation, IsAbstractOperationDif[]]
}
fun IsAbstractOperationDif(): set idOperation{
      {auxOp: idOperation|some absAux: absClassOperation,
      conAux:conClassOperation|
      (absAux.isAbstractOp!=conAux.isAbstractOp) and
      (absAux.nameOp=conAux.nameOp) and
      (absAux.nameOp= auxOp and conAux.nameOp= auxOp) }
}
pred ReturnOperation{
            isReturnOperation[]=True
}
fun isReturnOperation(): one Bool{
            #mapDiferentReturnOperation [ ]>0 => True else False
}
fun mapDiferentReturnOperation(): set Operation->Operation{
      {absAux:absClassOperation, conAux:conClassOperation |
      absAux.returnOp!=conAux.returnOp and absAux.nameOp=conAux.nameOp}
}
assert DiferentParameterOperation{
      all pmOp:isDiferentParameterOperation[],
      rul:r1d/isClassAbsINCon[]|pmOp=True and rul=True
}
fun isDiferentParameterOperation(): one Bool{
      (#conSameParameterOp[ ]>0) => True else False
}
fun conSameParameterOp(): set Operation {
      {con:conClassOperation|some absAux: absClassOperation,
      conAux:conClassOperation|
      (absAux.nameOp=conAux.nameOp and conAux.nameOp=con.nameOp) }
}
```

```
fun sameNameDiferentParameterOpCON(): set Operation {
      {c:conDifParameterOp|some c2:conDifParameterOp|
      c.nameOp=c2.nameOp and
      c.parameterOp.values.namePm!=c2.parameterOp.values.namePm}
}
fun conDifParameterOp(): set Operation {
      conClassOperation-  conSameParameterOp
}
fun mapParameterOp(): set Operation{
            sameNameDiferentParameterOpCON
}
assert ChangeProprietyOperation{
      all chOp:isChangeProprietyOperation[],
      rul:r1d/isClassAbsINCon[]|chOp=True and rul=True
}
fun isChangeProprietyOperation(): one Bool{
      ((r3IsAbsOp/isAbstractOperation []=True) or
      (r3ParamOp/isDiferentParameterOperation[]=True) or
      (r3RetOp/isReturnOperation[]=True) or
      (r3VisOp/isVisibilityOperation[]=True))
      => True else False
}
assert NewOperation{
            all nwOp:isNewOperation[], rul:r1d/isClassAbsINCon[]|nwOp=True
and rul=True
}
fun isNewOperation: one Bool{
      (#mapNewOperation>0)=> True else False
}
fun mapNewOperation: set Operation->Class{
      mapAllNewOperation->{c:conClass| some
c.operationsClass&mapAllNewOperation }
}
fun mapAllNewOperation: set Operation{
      idOperationTOperation[conClassOperation, difConAbs[]]
      +allOperationSameName
}
fun difConAbs: set idOperation{
                        (conClassOperationsName - absClassOperationsName)
}
fun allOperationSameName(): set Operation{
      {aux:conClassOperation| some con: difConAbs[]|
            aux.nameOp=con }
}
```

Categoria Relacionamentos

```
assert AssociaTOAgrega{
      all relAssTOAgr:isAssociationTOAgregation,
      rulAtr:r2InAtr/isAtributesAbsINCon[],
            rulClass:r1InClass/isClassAbsINCon[]|
            relAssTOAgr=True and rulAtr=True and rulClass=True
}
fun isAssociationTOAgregation: one Bool{
      (#assTranformAgre>0) => True else False
}
```

```
fun assTranformAgre: set Relation{
    idRelationAssTORelationAss[conRelation,{auxRel:conRelation.nameAss|
      some abs:absRelation, con:conRelation|
      abs.nameAss=con.nameAss and
      abs.nameAss=auxRel and con.nameAss=auxRel and
      abs.relationTypeAss=associa and con.relationTypeAss=agrega and
      abs.(start+end).class.nameClass =con.(start+end).class.nameClass and
      abs.(start+end).multiplicityAss =con.(start+end).multiplicityAss}]
}
fun assTranformAgreABS(): set Relation{
      { abs:absRelation|some tr:assTranformAgre[]| abs.nameAss=tr.nameAss}
}
fun assTranformAgreCON(): set Relation{
      { con:conRelation|some tr:assTranformAgre[]| con.nameAss=tr.nameAss}
}
fun mapAssTranformAgre: set Relation->idRelation{
      assTranformAgreABS->assTranformAgreABS.relationTypeAss +
      assTranformAgreCON->assTranformAgreCON.relationTypeAss
}
assert AssociaTOGeneraliza{
    isAssociationTOGeneralization=True and r2InAtr/isAtributesAbsINCon[]=True
            and r1InClass/isClassAbsINCon[]=True
}
fun isAssociationTOGeneralization: one Bool{
      (#assTranformGen>0) => True else False
}
fun assTranformGen: one Relation{
      {conAux:conRelation|some abs:absRelation, con:conRelation|
      con.nameGen=abs.nameAss and conAux.nameGen=con.nameGen and
      (con.superClass.nameClass=abs.start.class.nameClass) and
      (abs.relationTypeAss=associa and con.relationTypeGen=generaliza)}
}
fun assTranformGenABS(): set Relation{
      { abs:absRelation|some tr:assTranformGen[]| abs.nameAss=tr.nameGen}
}
fun assTranformGenCON(): set Relation{
      { con:conRelation|some tr:assTranformGen[]| con.nameGen=tr.nameGen}
}
fun mapAssTranformGen: set Relation->idRelation{
      assTranformGenABS->assTranformGenABS.relationTypeAss +
      assTranformGenCON->assTranformGenCON.relationTypeGen
}
assert ManyTOMany{
            all  mm:isManyTOMany, cl:clAbsINcon[],
rulAtr:r2InAtr/isAtributesAbsINCon[], rul:r1InClass/isClassAbsINCon[]|
            mm=True and cl=True and rul=True and rulAtr=True
}
fun isManyTOMany: one Bool{
      #mapAssociativeClassCon>=1 implies True else False
}
fun relManyTOManyAbs: set Relation{
      {rel:absRelation| rel.(start+end).multiplicityAss=muitos
      and rel.relationTypeAss=associa }
}

fun auxClassAssociation:set Class{
      idClassTOClass[conClass,relManyTOManyAbs.(start+end).class.nameClass-
            traceAbstractClass+traceConcreteClass]
}
```

```
fun clAbsINcon: one Bool{
      #{a:auxClassAssociation[], b:conClass| a in b}>0 => True else False
}
fun auxAssociationClassCon: set Class{
      associationClassConRHS[].(start+end).class&associationClassConLHS[].
      (start+end).class
}
fun associationClassConRHS: set Relation{
      {rel:conRelation|rel.relationTypeAss=associa and
      rel.start.multiplicityAss=muitos and rel.end.multiplicityAss=um }
}
fun associationClassConLHS: set Relation{
      {rel:conRelation|rel.relationTypeAss=associa and
rel.start.multiplicityAss=um and rel.end.multiplicityAss=muitos}
}
fun mapAssociativeClassCon: set Class->Relation{
            auxAssociationClassCon->associationClassConRHS +
            auxAssociationClassCon->associationClassConLHS
}
assert NewRelation{
      all newRel:isNewRelation, rulAtrb:r2InAtr/isAtributesAbsINCon[],
      rulClass:r1InClass/isClassAbsINCon[]|
      newRel=True and rulAtrb=True and rulClass=True
}
fun isNewRelation: one Bool{
      (#mapAllNEWRelations>0 )=> True else False
}
fun sameAssRelationProprietyAbsCon: set idName{
      {auxRel:conRelation.nameAss|some abs:absRelation, con:conRelation|
        abs.nameAss=con.nameAss and
        abs.nameAss=auxRel and con.nameAss=auxRel and
        abs.relationTypeAss=con.relationTypeAss and
        abs.(start+end).class.nameClass =con.(start+end).class.nameClass
       and abs.(start+end).multiplicityAss =con.(start+end).multiplicityAss}
}
fun sameAssRelationAbsCon: set idName{
      {auxRel:conRelation.nameAss|some abs:absRelation, con:conRelation|
            abs.nameAss=con.nameAss and
            abs.nameAss=auxRel and con.nameAss=auxRel and
            abs.relationTypeAss=con.relationTypeAss      }
}
fun sameGenRelationAbsCon: set idName{
      {auxRel:conRelation.nameGen|some abs:absRelation, con:conRelation|
            abs.nameGen=con.nameGen and
            abs.nameGen=auxRel and con.nameGen=auxRel and
            abs.relationTypeGen=con.relationTypeGen      }
}
fun allNewAssociation: set idName{
      isAssociation[conRelation].nameAss-isAssociation[absRelation].nameAss
}
fun allNewAgregation: set idName{
      isAgregation[conRelation].nameAss-isAgregation[absRelation].nameAss
}
fun allNewGeneralization: set idName{
      isGeneralization[conRelation].nameGen -
isGeneralization[absRelation].nameGen
}
fun isGeneralization(mod:Model.relationsModel): set Relation{
      {auxRel:mod| auxRel.relationTypeGen=generaliza}    }
```

```
fun isAssociation(mod:Model.relationsModel): set Relation{
      {auxRel:mod| auxRel.relationTypeAss=associa }
}
fun isAgregation(mod:Model.relationsModel): set Relation{
      {auxRel:mod| auxRel.relationTypeAss=agrega}
}

fun mapAllNEWRelations:set Relation{
    idRelationAssTORelationAss[conRelation, allNewAssociation]
    +idRelationAssTORelationAss[conRelation, allNewAgregation]
    +idRelationGenTORelationGen[conRelation, allNewGeneralization]
    + (idRelationAssTORelationAss[conRelation, sameAssRelationAbsCon]
    - idRelationAssTORelationAss[conRelation,sameAssRelationProprietyAbsCon])
}
```

ListRules

```
open RefinamentoDCAnaliseVerificacao/RuleRefinement/ListRules/ListRuleClass as l1
open RefinamentoDCAnaliseVerificacao/RuleRefinement/ListRules/ListRuleAtribute as l2
open RefinamentoDCAnaliseVerificacao/RuleRefinement/ListRules/ListRuleOperation as l3
open RefinamentoDCAnaliseVerificacao/RuleRefinement/ListRules/ListRuleRelation as l4
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule1/Rule1InclusionClass as r1a
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule1/Rule1NewClass as r1b
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule1/Rule1SameProprietyClass as r1c
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule1/Rule1ChangeProprietyClass as r1d
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule2/Rule2NewAtribute as r2a
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule2/Rule2ChangeProprietyAtribute as r2b
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule2/Rule2MoveAtributeAssociation as r2c
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule2/Rule2MoveAtributeGeneralization as r2d
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule2/Rule2TurnAtributeINTOClass as r2e
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule3/Rule3NewOperation as r3a
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule3/Rule3ChangeProprietyOperation as r3b
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule3/Rule3ParameterOperation as r3c
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule4/Rule4NewRelation as r4a
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule4/Rule4AssociationTOAgregation as r4b
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule4/Rule4AssociationTOGeneralization as r4c
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule4/Rule4MANYtoMANY as r4d

fun ListALLRules(): set idRule{
      l1/allRuleClasses + l2/allRuleAtributes + l3/allRuleOperations +
      l4/allRuleRelations
}
fun allRuleClasses(): set idRule{
      ruleNewClass + ruleChangeProprietyClass
}
fun ruleNewClass(): one idRule{
            (r1b/isNewClass)=True =>  {NewClass} else none
}
fun ruleChangeProprietyClass(): one idRule{
      (r1c/isSameProprietyClass=False) and (r1d/isChangeProprietyClass=True)
            =>  {ChangeProprietyClass} else none
}
fun allRuleAtributes(): set idRule{
      ruleNewAtribute + ruleChangeProprietyAtribute +
      ruleMovedAtributeAssociation+ ruleMovedAtributeGeneralization +
      ruleTurnAtributeINTOClass
}
fun ruleNewAtribute():one idRule{
      (r2a/isNewAtribute=True) =>{NewAtribute} else none
}
fun ruleChangeProprietyAtribute():one idRule{
   (r2b/isChangeProprietyAtribute=True) =>{ChangeProprietyAtribute} else none
}
```

```
fun ruleMovedAtributeAssociation():one idRule{
      (r2c/isMoveAtributeAssociation=True) =>
      {MovedAtributeAssociation} else none
}
fun ruleMovedAtributeGeneralization():one idRule{
      (r2d/isMovedAtributeGeneralization=True)
      =>{MovedAtributeGeneralization} else none
}
fun ruleTurnAtributeINTOClass():one idRule{
      (r2e/isTurnAtributeINTOClass=True) =>{TurnAtributeINTOClass} else none
}
fun allRuleOperations(): set idRule{
      ruleNewOperation +ruleChangeProprietyOperation +ruleParameterOperation
}
fun ruleNewOperation():one idRule{
      (r3a/isNewOperation=True) =>{NewOperation} else none
}
fun ruleChangeProprietyOperation():one idRule{
      (r3b/isChangeProprietyOperation=True)=>
      {ChangeProprietyOperation} else none
}

fun ruleParameterOperation():one idRule{
      (r3c/isDiferentParameterOperation=True) =>
      {DiferentParameterOperation} else none
}
fun allRuleRelations(): set idRule{
      ruleNewRelation + ruleAssociationTOAgregation + ruleMANYtoMANY +
      ruleAssociationTOGeneralization
}
fun ruleNewRelation():one idRule{
      (r4a/isNewRelation=True) =>{NewRelation} else none
}
fun ruleAssociationTOAgregation():one idRule{
      (r4b/isAssociationTOAgregation=True) =>
      {AssociationTOAgregation} else none
}
fun ruleAssociationTOGeneralization():one idRule{
      (r4c/isAssociationTOGeneralization=True) =>
      {AssociationTOGeneralization } else none
}
fun ruleMANYtoMANY():one idRule{
      (r4d/isManyTOMany =True) =>{MANYtoMANY } else none
}
```

MapResult

```
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule2/Rule2TypeAtribute as r2TypeAtr
open RefinamentoDCAnaliseVerificacao/ RuleRefinement/Rule2/Rule2VisibilityAtribute as r2VisAtr
open RefinamentoDCAnaliseVerificacao/RuleRefinement/ Rule2/Rule2DiferentNameAtribute as r2DifName
open RefinamentoDCAnaliseVerificacao/RuleRefinement/ Rule2/Rule2TurnAtributeINTOClass as r2AtrINTOClss
open RefinamentoDCAnaliseVerificacao/ RuleRefinement/Rule2/Rule2NewAtribute as r2NewAtr
open RefinamentoDCAnaliseVerificacao/RuleRefinement/ Rule2/Rule2MoveAtributeAssociation as r2MovAtrAss
open RefinamentoDCAnaliseVerificacao/ RuleRefinement/Rule2/Rule2MoveAtributeGeneralization as r2MovAtrGen
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule1/Rule1IsAbstractClass as r1IsAbsCls
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule1/Rule1VisibilityClass as r1VisCls
open RefinamentoDCAnaliseVerificacao/RuleRefinement/ Rule1/Rule1NewClass as r1NewCls
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule1/Rule1ChangeNameClass as r1NameCls
open RefinamentoDCAnaliseVerificacao/RuleRefinement/ Rule3/Rule3IsAbstractOperation as r3IsAbsOP
open RefinamentoDCAnaliseVerificacao/RuleRefinement/ Rule3/Rule3VisibilityOperation as r3VisOP
open RefinamentoDCAnaliseVerificacao/RuleRefinement/ Rule3/Rule3NewOperation as r3NewOP
open RefinamentoDCAnaliseVerificacao/RuleRefinement/ Rule3/Rule3SameNameOperation as r3NameOP
open RefinamentoDCAnaliseVerificacao/ RuleRefinement/Rule3/Rule3ReturnOperation as r3RetOP
open RefinamentoDCAnaliseVerificacao/ RuleRefinement/Rule3/Rule3ParameterOperation as r3ParamOP
```

```
open RefinamentoDCAnaliseVerificacao/ RuleRefinement/Rule4/Rule4AssociationTOGeneralization as
r4AssTOGen
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule4/Rule4AssociationTOAgregation as
r4AssTOAgr
open RefinamentoDCAnaliseVerificacao/RuleRefinement/Rule4/Rule4MANYtoMANY as r4mTOm
open RefinamentoDCAnaliseVerificacao/RuleRefinement/ Rule4/Rule4NewRelation as r4NewRel

fun mapTypeAtributeTransformations(): set Atribute->idType {
       r2TypeAtr/mapDiferentTypeAtribute }
fun mapVisibilityAtributeTransformations(): set Atribute->idVisibility{
       r2VisAtr/mapDiferentVisibilityAtribute
}
fun mapNameAtributeTransformations(): set Atribute ->Atribute{
       r2DifName/mapDiferentNameAtribute
}
fun mapAtributeINTOClassTransformations(): set Atribute->Class{
       r2AtrINTOClss/mapAtributeTurnINTOClass
}
fun mapNewAtributeTransformations(): set Atribute->Class{
       r2NewAtr/mapNewAtribute
}
fun mapMovedAssociationTransformations(): set Class->Atribute{
       r2MovAtrAss/mapMovedAtributeClass
}
fun mapMovedGeneralitionTransformations(): set Class->Atribute{
       r2MovAtrGen/mapMovSuperAtribute
}
fun mapDiferentNameClassTransformations(): set Class->Class{
       r1NameCls/mapClassSameIdDiferentName
}
fun mapNewClassTransformations(): set Class{
       r1NewCls/mapNewClass
}
fun mapVisibilityClassTransformations(): set Class->idVisibility{
       r1VisCls/mapDiferentVisibilityClass
}
fun mapIsAbstractClassTransformations(): set  Class->Bool{
       r1IsAbsCls/mapIsAbstractClass
}
fun mapIsAbstractOperationTransformations(): set Operation->Bool{
       r3IsAbsOP/mapIsAbstractOperation
}
fun mapNewOperationTransformations(): set Operation->Class{
       r3NewOP/mapNewOperation
}
fun mapParameterOpTransformations(): set Operation{
       r3ParamOP/mapParameterOp
}
fun mapReturnOpTransformations(): set Operation->Operation{
       r3RetOP/mapDiferentReturnOperation
}
fun mapVisibilityOpTransformations(): set Operation->idVisibility{
       r3VisOP/mapVisibilityOperation
}
fun mapNewRelationTransformations(): set Relation{
       r4NewRel/mapAllNEWRelations
}
fun mapAssociationTOGeneralizationTransformations(): set Relation->
idRelation {
       r4AssTOGen/mapAssTranformGen
}
```

```
fun mapAssociationTOAgregationTransformations(): set Relation->idRelation{
      r4AssTOAgr/mapAssTranformAgre
}
fun mapMANYtoMANYTransformations(): set Class->Relation{
      r4mTOm/mapAssociativeClassCon
}
fun typeRefinement(): one idRule{
            (#absClassName= #conClassName and
            #absRelationAssociationName=#conRelationAssociationName and
            #absRelationGeneralizationName=#conRelationGeneralizationName)
            =>  {identidadeRefina} else {CONRefinaABS}
}
```